THE DESIGN OF A FIRST COURSE IN PROGRAMMING

by

Michael P. Brady B.Sc., H.D.E.

A Master's Thesis

Submitted in partial fulfilment of
the requirements for the award of
M.Phil.
of the Loughborough University of Technology

MAY 1986

Supervisor : Professor A.C. Bajpai, Director of C.A.M.E.T. and Head of the Department of Engineering Mathematics, Loughborough University of Technology.

Local Supervisor : Dr. John O'Donoghue, Head of Mathematics, Thomond College, Limerick, Ireland.

The Design of a First Course in Programming — Michael P. Brady

THESIS ABSTRACT

A course was designed to teach Top-Down programming to second level students who had no previous computer experience. The purposes of the course were a) to enable them to become computer literate and b) to develop their problem-solving ability. The course was designed to teach programming in a manner which was independent of any particular programming language or machine. This approach was prompted by dissatisfaction with traditional courses which generally concentrate on the syntax and semantics of a particular programming language, at the expense of developing important underlying concepts.

Initially, a review of the history of programming languages was carried out to identify the essential elements of programming. This review found that there was general agreement about the fundamental importance of structure and that it was not necessary to use all of the control constructs contained in the available languages (BASIC, COMAL and PASCAL).

Both a mini-language, containing just two control structures, and a diagrammatic representation (structure diagrams) of the mini-language were then designed. The chosen control structures were IF/THEN/ELIF/ELSE for selection and a WHILE loop for iteration. The students were trained to solve problems using the mini-language and structure diagrams and were supplied with translation rules to convert their solutions into COMAL. Translation rules were also drawn up for PASCAL and BASIC.

The course was tested with girls aged 15 and 16 years in a Dublin secondary school. These trials showed that the method may be used successfully with students of this age.

The Design of a First Course in Programming

Michael P. Brady

KEYWORDS

Top-Down Analysis

Structured Programming

Control Structures

Mini-Language

Structure Diagrams

Problem-Solving

Second Level Students

ACKNOWLEDGMENTS
==========

I wish to express my gratitude to Professor A.C. Bajpai for providing me with this research opportunity and for his support. I also wish to thank my local supervisor, Dr. John O'Donoghue of Thomond College Limerick, for his practical advice and for his constant interest in the project.

I wish to thank the staff of the Holy Faith Secondary School, The Coombe, Dublin for their co-operation. In particular I would like to thank Sister Aideen, who made the school's computer facilities available to me at all times, Mia Delaney, who provided much background information, and Christina Nulty, who offered many helpful suggestions in preparing the manuscript.

I would especially like to thank Marita McGrath who helped teach the course and who offered much constructive advice and support.

Finally, I would like to thank my wife, Bridin, whose constant support and encouragement were invaluable.

TABLE OF CONTENTS
==========

CHAPTER 1

INTRODUCTION

A recent survey of Irish schools [1] concluded that many of our school leavers are computer illiterate. This is clearly undesirable in a society which is increasingly influenced by information technology and in which the rate of technological innovation is so rapid. The need for familiarisation with the new technology as part of the curriculum is now generally recognised. The government has expressed the hope that encouragement will be given to the promotion of the study of computers as part of the school curriculum [2] and has begun this process by including Computer Studies modules in both the Intermediate and Leaving Certificate programmes and by arranging for the supply of computing equipment to all second level schools. This view, that education should embrace the new technologies, has been endorsed by the recently formed Curriculum and Examinations Board [3] and by many other bodies representing teachers at all levels of education.

The broad goal of all computer education is to provide students with an understanding of the operation and applications of computers and to alert them to the limitations and difficulties associated with the new technology. This may be achieved in a number of different ways including Computer Studies as a subject in its own right, computer aided learning and the use of computers as

tools in various subject areas. One overall aim of all such modes of computer use is that students should develop an understanding of computers and become competent in their use. The achievement of computer literacy may also involve discussion of issues such as the social impact of computers, the history of computing, the rudiments of computer hardware and the use of various software packages.

Computer literacy, however, must also involve some understanding of programming, which is an integral part of all computing. This is not to suggest that our students should be trained as professional programmers but that understanding of computing could not be complete without some awareness of how programs are designed. If this awareness is not achieved, then real computer literacy is not possible as the student could not really appreciate how the machine is totally dependent on the skill of the programmer and the accuracy of the supplied data.

Learning how to program can be an enjoyable, stimulating and highly rewarding educational experience. It is a creative activity requiring the integration of many skills and requires much more active involvement on the part of the student than the use of software packages. When programming, the student is in control of the machine and not simply responding to the machine's instuctions which is the case in using many software packages. Papert [4] has warned of the danger of allowing the machine to program the student in such circumstances. Programming, therefore, is likely to help the student to develop a sense of mastery and power in

relation to computers which may not be achieved by any other computer related activity. Thus the affective benefits of learning to program may be important in the development of a positive attitude towards computers and technology in general.

The importance of programming is recognised by the majority of schools and Computer Studies (which normally includes a large element of programming) is by far the chief mode of computer use in schools [1]. Its importance is also recognised by the Department of Education which has suggested that programming should be a major element of the current second level computer modules at both junior and senior cycle [5,6]. This emphasis on programming is further underlined by the Department of Education's publication of a book on programming as an aid to the teaching of Computer Studies [7] and by the fact that most of the software supplied by the Department to schools consists of programming languages (PASCAL, COMAL, LOGO and MICRO-PROLOG).

In addition to its contribution to the achievement of computer literacy, learning how to program may also lead to an improvement in general problem-solving skills. While it is recognised that problem-solving activity can be generated in all academic disciplines, recent developments in computer science have produced new methodologies which explicitly focus on the problem-solving process. Using these techniques, it is possible to teach students how to approach problems in a systematic and disciplined way which might

then be applied to a broad range of situations. The relevant programming skills, which include planning, problem decomposition, anticipation of outcomes, recognition of relationships, testing, revising and persevering, are applicable to many non-computer problem-solving situations. When programming, students develop and experiment with their own hypotheses, criticise their own work constructively and must persevere until their solution is fully correct. The emphasis is firmly placed on the process of solving the problem rather than on the product (the answer). The actual use of a computer for testing programs supplies immediate and valuable feedback which would be unthinkable in most other problem-solving situations.

From the students' viewpoint, problem-solving situations are very often frustrating and time consuming. Hativa [8] has found that undergraduate students generally do not see the teacher's role as that of a 'skill developer' but rather as a 'transmitter' of knowledge, whose main function is to impart facts. A necessary corollary of this is that students see themselves as passive 'receivers' of information and may resent being asked to solve awkward problems. It is likely that this point of view is even more prevalent in second level schools due to the increasing pressure of public examinations for which the acquisition of facts is very important. In most other subjects on the curriculum, second level students will have already encountered an enormous number of facts and techniques that must be remembered and understood before they can be applied

to problems. In many cases these basic techniques are not well comprehended and so the learner is not in a position to solve problems. This difficulty can be overcome in a first programming course as the students will be starting a completely new subject in which there is no previously learnt body of knowledge to be remembered.

In a first programming course, non-transferable, low-level knowledge should be kept to a minimum so as to allow the learner to concentrate on the essential problem-solving techniques at the heart of programming. This suggests that a full programming language should not be used because of the large amount of time required to learn all the syntax and semantics of the language and because a lot of such knowledge is not transferable to other programming languages, or even to a different implementation of the same language. More importantly, the need to learn all the petty rules and regulations of a particular programming language would distract attention from the problem-solving process and might well defeat the whole purpose of a course intended to develop high level cognitive skills. Dissatisfaction was felt with the current practice in teaching introductory programming courses which invariably seemed to concentrate on the details of whatever programming language was most easily implemented on the available hardware. It was felt that the teaching of programming should not be dependent on the available hardware as this was self defeating due to the very wide range of languages, the numerous versions of each language and the rapid changes taking place in this area.

Thus concentration on the most readily available language could well lead to students being conversant with the minute details of some language which was obsolete before they left school and this was not desirable.

With this in mind a course was developed which attempted to develop the essential programming skills but which avoided, as far as possible, concern with the specific details of any one programming language. This was done by designing a mini-language which contained only the essential programming structures and which was considered suitable for students in the senior cycle of our second level schools. These essential programming structures were isolated by a study of the history of programming languages over the past thirty years including the development of the structured programming movement. This is outlined in Chapter 2. This restriction of the programming language should not be considered an impediment to 'proper' programming but rather as an aid to structured programming. Wells [9] has concluded that in spite of the proliferation of programming languages, general purpose languages are converging towards 'everyday' algorithmic language with relatively few essential constructs. It has also been claimed by Dijkstra [10] that the use of a small, elegant language can actually help programmers to find algorithms that are much more difficult to find using more extensive languages. The educational importance of structured programming is discussed in Chapter 3.

It was recognised that some diagrammatic representation of the programming structures was required but an examination of the current methods used in most textbooks, described in Chapter 5, found that none of the available methods was desirable or suitable. A new diagrammatic representation was then designed to match the chosen programming structures and to represent the hierarchical nature of programs clearly and unambiguously. Once a solution has been designed diagrammatically, it is vital to be able to convert it into an implementable programming language easily and directly. Sets of translation rules were therefore formulated to convert the diagrammatic solutions into the various popular programming languages. The mini-language, its diagrammatic representation and the translation rules are described in Chapter 4.

Thus a complete system was developed which allowed the students to concentrate on the design of solutions rather than on the syntax of a programming language. It provided a clear hierarchical diagrammatic representation of completed programs which allowed for the convenient translation of the solution into any of the popular programming languages. In short, a system was developed which allowed students to concentrate on solving problems and which relegated the actual implementation of solutions on a computer to a simple 'automatic' clerical task.

Avital and Shettleworth [11] have suggested that the key to fostering higher level abilities is exposure and Polya [12] has claimed that solving problems is a practical skill which

is acquired by imitation and practice. This means that if students are to learn how to write good programs then they must be shown examples of good programming and must also be given a broad range of stimulating and interesting problems on which to work. To maintain motivation, students must also feel that they are successful and this is best achieved by ensuring that the problems they are given are within their capabilities. The general strategy followed was to introduce each new topic by means of a problem or a set of problems. Solutions for these were provided by the teacher in a step-by-step fashion, emphasising the reasons for each step rather than presenting the solution as a finished product. The students were then given sets of similar problems to solve, both in class and for homework. While many of these problems required fairly straightforward application of the new technique, some were sufficiently remote from the original examples to require higher level cognitive activity. Thus the course was based around a large set of problems which were devised (or in some cases adapted from textbooks) to illustrate widely applicable programming techniques and structures. Much emphasis was placed on students' homework which normally consisted of solving problems diagrammatically. These solutions were then usually implemented on the school's computers.

The students' repertoire of programming techniques was built up throughout the course and towards the end they were given moderately large problems to solve over a period of a few weeks. All the techniques necessary to solve these problems

had been encountered during the course but because these problems were larger than those previously encountered they required more analysis and more commitment on the part of the students. That they were very successful at these problems was a good indication of the effectiveness of the course. The course, its implementation and its evaluation are described in Chapters 6, 7 and 8 respectively. Conclusions are drawn in Chapter 9.

CHAPTER 2

HISTORICAL BACKGROUND

2.1 INTRODUCTION

In the past thirty years the nature of computer programming has changed dramatically. It is intended to review the major developments that have taken place in relation to programming and programming languages and their uses in education. It is not intended to present a comprehensive study of the major programming languages but to review the important trends with a view to isolating what is essential, what should be taught and how it should be taught.

2.2 THE 1950s.

In the early fifties all programming was done in machine code or assembly language [13]. Programming was considered to be a very complex and highly creative art and the skills of the early programmers were directed towards overcoming the limitations of the available hardware. These limitations included the absence of floating point calculations, slow processing speeds, very small memories, and restricted instruction sets. Programs had to be written to fit into the tiny memories and to run as quickly as possible to make the most efficient use of the large, very expensive computers. Programmers were prized for their ability to write tricky, efficient code which took advantage of the particular idiosyncrasies of the machine that they were using.

The mid fifties saw the beginning of the evolution of higher level 'algebraic' languages e.g. FLOWMATIC (1955) and MATH-MATIC (1957). These languages freed programmers from many of the 'red-tape' issues of programming by allowing them to write more sentence-oriented code, to use decimal numbers and to avail of subroutine libraries. In effect, they allowed programmers to concentrate more on algorithmic issues and less on machine issues. Initially, these new languages met with a lot of resistance in the computing community, mainly due to the fact that they slowed processing speeds by a factor of about ten [13].

Towards the end of the decade there was an enormous effort put into the development of high level languages. It began with the recognition of a basic economic problem, i.e. that programming and debugging costs were excessive and were growing all the time. The response was to try to develop languages which made programming easier, which were machine-independent and which were problem-oriented. The three most important languages introduced in this period were FORTRAN, COBOL and ALGOL '60.

FORTRAN was intended for use by scientists and engineers for mathematical computation. Among the language concepts introduced by FORTRAN were [14]:

1. Variables and expressions (arithmetic and boolean).
2. Arrays (whose dimensions were known at compile time).
3. Iterative and conditional branching control structures.
4. Programs as sets of subroutine or function segments

that could be compiled independently.

The committee that designed FORTRAN was unaware of many of the issues of language design that were later to become important such as block structure, type declarations etc.[13]. In fact, it considered the design of the language to be a rather trivial prelude to the 'real' problem which was the design of the compiler. As a result, the language has subsequently been described by Dijkstra as an efficient coding system but one with few conceptual aids to assist the programmer [15].

COBOL was designed for business data processing, in which the emphasis is on file-handling, with relatively little 'computing'. The designers intended that the language could be used by novice progammers. It was also hoped that, by keeping it as close to natural language as possible, it could be read and understood by management [16]. Hence, much attention was given to making it easy to read, as opposed to making it easy to write or to learn. Some important concepts introduced by COBOL were [14]:

1. The IF/THEN/ELSE structure.
2. Separation of procedural statements from data description.
3. Natural language style.
4. Record data structures.

FORTRAN and COBOL were immediately successful and were very widely implemented. Even today they are among the most widely used languages and most in-house company training courses still use COBOL [17]. The reasons for their great success and

continued use are:

1. COBOL was chosen as the required language on all U.S. Department of Defence computers.
2. There was a huge investment in applications programs in both languages.
3. They became standardised.

The main impetus for the design of ALGOL '60 came from a desire to allow algorithms to be written more clearly and conveniently. It was intended mainly for scientific computation. The language's defining document [18] introduced a method of language definition, Backus-Naur Form (BNF), which was almost as important as the language itself. This was a huge advance on previous language definition techniques and initiated the idea of language as an object of study in its own right, rather than as a tool for facilitating the specification of programs. Some of the new concepts introduced in ALGOL '60 were [14]:

1. Block structure.
2. Explicit type declarations for variables.
3. Scope rules for local variables.
4. Nested IF/THEN/ELSE statements.
5. Call-by-value and call-by-reference parameters.
6. Recursive subroutines.
7. Dynamic arrays.

However, the most important elements of ALGOL '60 were its sense of simplicity and its conciseness. Although ALGOL '60 was never widely implemented, its style has been much more influential in subsequent language design than either FORTRAN

or COBOL which have had very little impact in this area.

2.3 THE 1960s.

By 1960 the debate concerning the use of high level languages was over. Machine coding had become the exception rather than the rule. During the sixties very many new languages were invented and by the end of the decade there were about 170 languages in use in the U.S. alone [19]. Approximately half of these were designed for special purposes such as string processing and pattern matching (SNOBOL), simulation (SIMULA, SIMSCRIPT), education (BASIC, APL), while the rest, including PL/1 and ALGOL '68, were general purpose languages.

In the early sixties programming was considered to consist mainly of coding in some particular language. The notion was common that scientific and commercial programmers, who used different languages, should be trained separately. Not only were languages designed for each group but each group had its own computers [20]. As the available hardware became more powerful the distinctions began to fade. The files that scientists were processing were as big as those in commercial installations, while commercial users were beginning to perform linear regressions and factor analyses on market data. Companies began to object to buying two sets of hardware and employing and educating two sets of programmers.

This, combined with the general belief at that time that programming was simple as long as the language was powerful enough and the computer fast enough, led to the development of

large powerful languages which attempted to combine the features of scientific and commercial languages. PL/1 was one such language which was developed from FORTRAN, COBOL and ALGOL. The idea of separately compiled subroutines sharing common data was taken from FORTRAN. Data structures were taken from COBOL. Block structure and control constructs were taken from ALGOL. PL/1 illustrates the advantages and problems associated with large general-purpose languages. The power and richness of such languages lead to complexity in both language definition and language use, making verifiability and readability of programs a major problem [14].

2.4 STRUCTURED PROGRAMMING.

By the end of the sixties the enormous advances in hardware design and the corresponding decrease in hardware costs meant that software had become by far the most expensive part of a computer system. It was also acknowledged that there were very serious problems in the area of software development. Most software projects were taking longer to complete and costing more than planned. Worse still, the end product was very often unreliable. It was realised that programming was a difficult task and that fast machines and powerful computer languages did not make it much easier. Prior to this, large software projects had been designed to minimise development costs rather than total costs for the lifetime of the piece of software. This led to a disproportionate emphasis on achieving speedy implementation and a corresponding neglect of both the initial problem specification and the production phase of the program, which might involve frequent modifications. It was estimated

that 70% of programmers' time was devoted to the maintenance of existing programs [21]. Maintenance and revision of software requires that someone other than the author is capable of understanding the original design. This was very often not the case and large amounts of time were being spent analysing existing code. Even at the development stage more time was being spent debugging than on algorithmic design.

The search for programming improvements led to an analysis of the fundamental structures of programming and a greater emphasis on the methodology of problem-solving. Research was directed away from the development of powerful new languages towards control of the complexity, cost and reliability of large programs. The new methodologies developed at this time are usually grouped under the heading of structured programming. Structured programming was originally considered to be programming without GOTO statements, substituting clearer control structures instead. It has since come to mean designing programs in such a way that they are simple, verifiable, reliable and maintainable. It is more concerned with using the programmer's time efficiently than with machine efficiency.

The first example of a structured language, PASCAL, appeared in the early seventies. PASCAL was designed by Niklaus Wirth who was dissatisfied with the major languages because of their over-elaborate constructs, which were difficult to explain logically and convincingly and which often defied systematic reasoning [22]. PASCAL was a return to a smaller, simpler type of language, based on the style of ALGOL '60 but providing

richer data structures and programmer-defined type specifications. It could also detect many programming errors as syntax errors because of its built-in protection against both improper mixing of types and the assignment of illegal values to variables. Because of its conceptual simplicity, it has been possible to develop a complete formal definition of the language [23]. The existence of this formal definition has in turn led to the widespread use of PASCAL as a base language for program verification research.

The techniques of structured programming (which are discussed in Chapter 3) have had a great impact on both academic Computer Science and on production programming. During the seventies, facilities for structuring were added to most of the major programming languages. However, correctness proofs, which are the more formal aspect of structured programming and which are of great interest to computer science researchers, have had practically no effect on general commercial programming. In order that a program may be proved correct, it must be developed from control structures that are well understood and this is a strong argument in favour of very modest, very systematic programming languages.

So after more than a decade of debate, academic computer scientists are promoting the use of simpler, more systematic and safer languages but FORTRAN, COBOL and BASIC (albeit with structuring facilities added in many cases) are still generally favoured by commercial users and even larger languages such as ADA are being designed. This situation has

been summed up by Wells [9], who has observed that good progress has been made in the area of language design, but that progress has been less dramatic with respect to the use of well designed languages.

2.5 COMPUTERS IN EDUCATION.

By 1962 computers were beginning to appear in third level institutions and were being used mainly by students of Science, Engineering and Mathematics [24]. In the early sixties there were no Computer Science departments in universities and most students who studied computers did so in short courses organised by Maths/Science departments. These courses were usually geared towards programming in FORTRAN.

One attempt to overcome this Maths/Science bias was the development of BASIC at Dartmouth College in the early sixties [25]. Its development was motivated by the need to make it possible for non-experts to program without committing themselves to a large amount of preliminary study. As about 75% of the students at Dartmouth were non-science majors, the group which produces most of the decision-makers in business and government, it was considered important that they should have some knowledge of computing to help them make sensible decisions about computers in their subsequent professional lives. Access to computing prior to this had involved punched cards and both intellectual and administrative hurdles. The designers of BASIC sought to overcome these by making it an on-line, simple to learn, interactive language. This had now become possible due to the arrival of time-sharing and cheap teletext terminals.

The language itself was a modification of FORTRAN. It was never intended that it should be used for solving very large or difficult problems - FORTRAN was supposed to be used for these. BASIC was very successful because of its:

1. Simple syntax.
2. Easy operating system.
3. Cheap implementation.
4. Interactive nature.

Despite its success, BASIC was developed at a time when the nature of programming was undergoing great changes and it embodied structures and practices which were later seriously questioned.

By the end of the sixties it was generally accepted that computing was a science in its own right and Computer Science departments began to appear in many universities. In 1970 it was estimated that there were about 300 college degree programmes in Computer Science in the United States [26]. The need for discipline in programming technique was beginning to be recognised at this time. The curriculum committee of the Association for Computing Machinery (ACM) has issued recommendations on the content of college Computer Science courses at regular intervals since the mid sixties. In general, these recommendations emphasise systematic algorithm development and clear programming style. A 1984 document, concerned with introductory College Computer Science courses, strongly emphasises Top-Down design and stepwise refinement [27]. Wirth designed PASCAL mainly as a medium for teaching programming in a systematic manner. He maintained that the

language in which programming is taught profoundly influences subsequent habits of thought, and that the concepts embodied in the language feed back into the learner's style of problem analysis, influencing the way in which problem-solving skills develop. This would indicate that great care should be exercised in the choice of language to be used for introductory programming courses.

PASCAL has been very widely accepted by the third level academic community and is currently used in most university Computer Science courses. Up to now it has not been available on many microcomputers and so has not been widely used in second level schools.

Computing in second level schools began in the late sixties. As very few schools could afford to buy computing equipment at this time, those schools which were involved had to beg and borrow computer time from local commercial firms and from universities. Programs were generally prepared on punched cards or paper tape and brought to the computer installation in the evenings. The nature of the work done under these very difficult circumstances depended entirely on the facilities available rather than on any clearly defined educational objectives. It was recognised, however, that the design of algorithms was of fundamental importance and that high level languages should be used wherever possible [26].

Towards the end of the seventies the arrival of affordable microcomputers caused dramatic changes in educational computing. Schools were now in a position to buy their own

hardware. The demand for computer courses suddenly increased and schools that had no experience or expertise in computing came under pressure to provide computer courses. When these first micros appeared there was very little educational software available and so teaching programming was the only viable educational activity. As they had very restricted memories, BASIC was the only possible programming language and so learning BASIC became the norm for teacher in-service courses. Consequently, many teachers' first experience of computing was in the use of BASIC. Such teachers and school administrators, who were unaware of the whole structured programming debate, were then forced into making important decisions concerning the choice of both hardware and programming languages.

This has, unfortunately, led to the entrenchment of BASIC. This trend is reinforced by the popular computer magazines, many of which heavily emphasise programming in BASIC. The hardware manufacturers compound this problem by continuing to supply primitive versions of BASIC with their machines. This is in spite of the fact that the original reason for using BASIC (i.e. tiny memories) is no longer valid. The argument most often used to justify the choice of BASIC is based on its widespread availability and the fact that much software is written in it. This, together with the fact that the language most widely taught is going to be thereafter the one most widely used, is, according to Wirth [22], "the safest recipe for stagnation in a subject of such profound pedagogical influence".

The inadequacy of BASIC as a language for learning programming is recognised by many educators. Efforts to overcome the problems associated with BASIC have included the promotion of LOGO, COMAL and the so-called structured BASICs. LOGO has received widespread attention in the primary and junior secondary sector. COMAL has been adopted by many second level schools in Denmark, Ireland, Sweden, Scotland and more recently in the United States [28]. A version of BASIC which is almost identical to COMAL has recently been released for the RML 380Z computer [29]. There is also a 'structured' version of BASIC available for the B.B.C. computer. As both of these machines are widely used in British schools, these developments are welcome.

One recent decision which may have a profound effect on the use of BASIC in American schools is the selection of PASCAL as the sole language to be used in the Advanced Placement Computer Science Examination [30]. This is an examination taken at the end of second level schooling by students who wish to pursue Computer Science courses at third level. As there are currently 300,000 students enrolled in introductory, third level, Computer Science courses in the United States [30], the demand for courses leading to the Advanced Placement Examination is certain to be great. This demand will have to be met by supplying courses in PASCAL at second level which should, in turn, stimulate manufacturers to supply versions of PASCAL for the popular school machines.

2.6 THE IRISH SITUATION.

Irish second level schools began using computers in the early seventies. Much of the impetus for the use of computers in schools has come from the Computer Education Society of Ireland (C.E.S.I.) which was founded in 1973 by Professor A.C. Bajpai and which has since provided the main forum for discussion and formulation of ideas in relation to computer education. The first Department of Education computer training course for teachers was held in 1970. Since then, numerous courses have been sponsored by the Department of Education, often in conjunction with C.E.S.I. These are usually one week courses and are held during the school holiday periods. In addition, other institutions (universities, teachers' centres etc.) have offered computing courses for teachers. None of these courses (including the Department of Education's) leads to an officially recognised teaching qualification.

Many of the teachers who have taken part in these courses are now taking their students through the computing option on the Leaving Certificate Mathematics course. This option, begun in 1981, was, up to 1984, the only recognised computer module in our second level schools. The module, intended as an interim arrangement pending the introduction of Computer Studies as a full subject on the curriculum, consists of approximately 35 hours instruction and was adopted by about 200 schools in its second year of operation [31]. In 1984, a similar module was introduced for junior cycle pupils in second level schools. There is no written examination of either module and both syllabii [5,6] leave much to the discretion of the teacher.

Therefore it is not clear what exactly is being taught but it would seem that in most cases the tendency is to concentrate on programming [32].

So, up to now, the principal use of computers in schools has been to teach programming. This has stimulated much debate about the choice of a suitable programming language for schools. Although there were earlier attempts to introduce low level languages [33], it seems to be fairly well accepted now that what is needed is a high level language. The choice, therefore, is between BASIC, LOGO and COMAL as no other languages are widely available as yet.

Of these three, LOGO has only recently become widely available but is already very popular in primary schools. It is likely that its use will increase dramatically in second level schools in the near future. BASIC, of course, is available for every micro (at no extra cost) and so is very widely used in second level schools. In 1981, the Department of Education drew up an , internal report on the use of computers in schools which has not been published [34]. It would seem, however, from subsequent actions by the Department that decisions were taken to promote COMAL, adopt the Apple II computer and postpone indefinitely the introduction of Computer Studies as a full subject.

The decision to promote a practically unknown but very well structured language, COMAL, was a courageous one and was generally welcomed by those who were already involved in computer education, including C.E.S.I. However, the

implementation of COMAL has caused many problems. The Department of Education has assisted all second level schools to buy one Apple computer. Each of these machines is capable of running COMAL but if schools buy extra Apple machines they are unable to implement COMAL on them without fitting a Z-80 card in each new machine. As many schools are now buying networks of from five to ten machines this large extra expense may tempt them to revert to BASIC. Similarly, as COMAL is supplied in ROM form for the B.B.C. computer it is necessary to buy one for each machine and this may be prohibitive if a large number of machines is to be bought. There is , however, a very cheap version available for the Commodore 64 which may be run on networks at no extra cost. So while the choice of COMAL must be applauded, the high cost of implementing it has tended to prevent its universal use in our schools. At the moment it seems that schools' programming courses are fairly evenly divided between BASIC and COMAL.

It is unfortunate that the introduction of computing into our schools has not been guided by a coherent overall plan. In the past, schools have been happy to accept any facilities that they could get without really examining their educational needs, but the falling cost of equipment suggests that schools will soon be able to pick and choose between hardware, software, and programming languages. It is in this context that there is now a great need to clarify the issues concerning computing in schools. The teaching of programming is one such major issue which is of fundamental educational importance and which requires serious debate. Programming

courses in our schools are important not just because of the need to produce well trained computer specialists to support our industry and economy but because they can make a major contribution to the achievement of computer literacy and can help towards the development of general high level cognitive skills. This can best be achieved if the subject is approached in a systematic way and if modern techniques of structured programming are adopted.

If this is to be done then it is necessary to examine in detail the techniques of structured programming and to devise a way of presenting them which is suitable for students at second level.

CHAPTER 3

EDUCATIONAL ROLE OF STRUCTURED PROGRAMMING

3.1 INTRODUCTION

Schools are at present coming under increasing pressure from both parents and students to provide courses in Computer Science. Many schools are responding to this pressure by acquiring computer equipment and providing courses in programming but in many cases insufficient thought and preparation has been put into the organisation of these courses. There is now an urgent need to examine the role of programming in second level schools and to decide on how it should be taught.

The teaching of programming is important because if we are concerned with educating our pupils for maturity and to accept responsibility for their actions as adults, then we must be concerned with their ability to think reasonably and logically and to act accordingly. This ability, to formulate a reasonable plan of action and to carry it through, is the very essence of programming. The study of programming in a systematic way can be a powerful facilitator for the development of higher level skills and abilities, encouraging an algorithmic, procedural approach to problem-solving. The unambiguous semantics of a programming language together with the precision of syntax required, can give our students insight into the power of language and the care required to communicate clearly. Moreover, these skills, if developed

carefully, are more widely applicable and are more permanent than mere knowledge of specifics, and so are likely to benefit our students' performance in other subject areas.

While we are not in the business of producing professional programmers, we can learn from the developments that have taken place in professional programming practice and academic computer science during the past twenty years. The main lesson that has been learnt is that good programming is structured programming and so, before developing an introductory course, it is necessary to review just what is meant by structured programming.

3.2 STRUCTURED PROGRAMMING

The principal concern of structured programming is that programs should be designed and coded in such a way as to make them correct and easily understood. Structured programming grew out of a recognition of the limitations of the human mind when confronted by large problems which may involve very many details. It is an attempt to limit the complexity with which the programmer has to deal at any one time. The methods used to achieve these objectives are:

1. Careful definition of the problem to be solved.
2. Design of solutions in a Top-Down manner.
3. Use of a limited number of carefully chosen control structures.
4. Use of a clear and consistent coding style.

3.2.1 PROBLEM DEFINITION

This involves developing and clarifying the exact specifications of the problem. It may be a fairly lengthy process as many problems are poorly defined and ambiguous initially. However, it is essential that due consideration be given to this part of the programming process, as no problem can be solved unless it is well understood. Some of the issues that should be examined at this stage are:

1. In what form will the data be supplied?
2. Within what reasonable limits is data expected to be?
3. What errors should be anticipated and what action should be taken if an error is found?
4. How will the end of the data be signalled?
5. May the input values be discarded after they are used in the computation?
6. What should be done if some operation cannot be successfully completed?
7. In what form and to what degree of accuracy should output be provided?
8. Is there any indication of the amount of output that will be produced?
9. What changes in the problem statement are likely to occur during the lifetime of the program?

3.2.2 TOP-DOWN DESIGN

The structured programming debate, at least as it has taken place in relation to education, has been mainly concerned with the choice of programming languages and control structures. This is a pity, as it is more important that consideration be

given to the overall structure of programs. Polya has suggested that in solving complex problems it is possible to distinguish between 'great' steps and 'small' steps [12]. In finding a solution it is important to organise the great steps first. Wirth has proposed a method of 'stepwise refinement' by which a large problem is broken down into a few sub-problems ('great' steps) [35]. These sub-problems are then continuously refined if necessary, as in figure 3.1, until each sub-problem may be considered to be trivial.

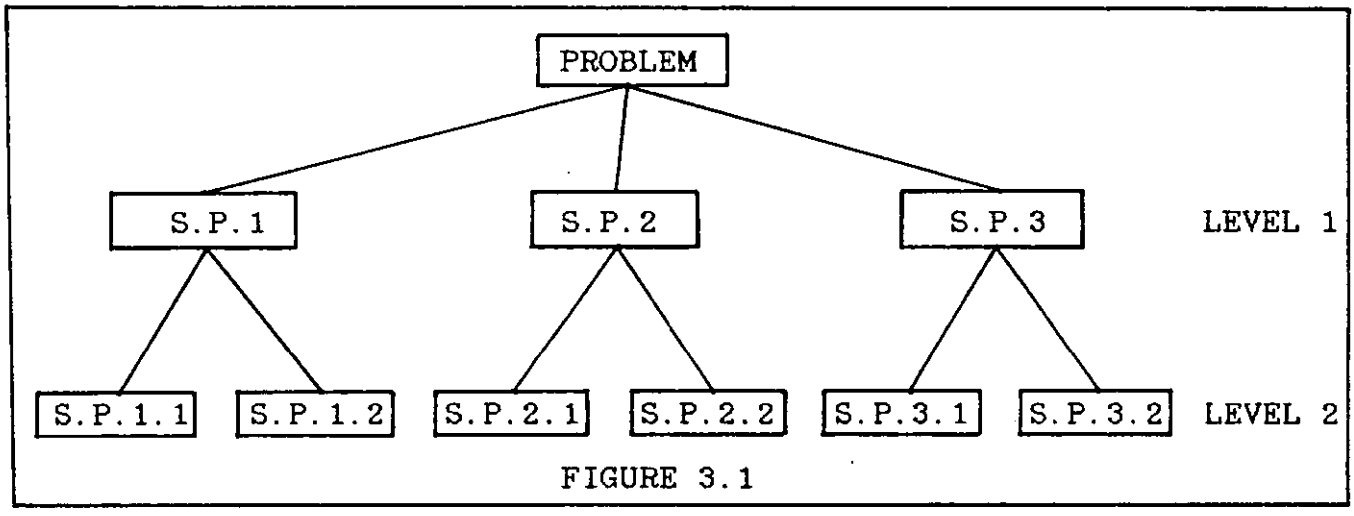


FIGURE 3.1

It is usual to start with a high level, abstract program. The level of abstraction is chosen to make the program short, understandable and obviously correct [36], as in figure 3.2.

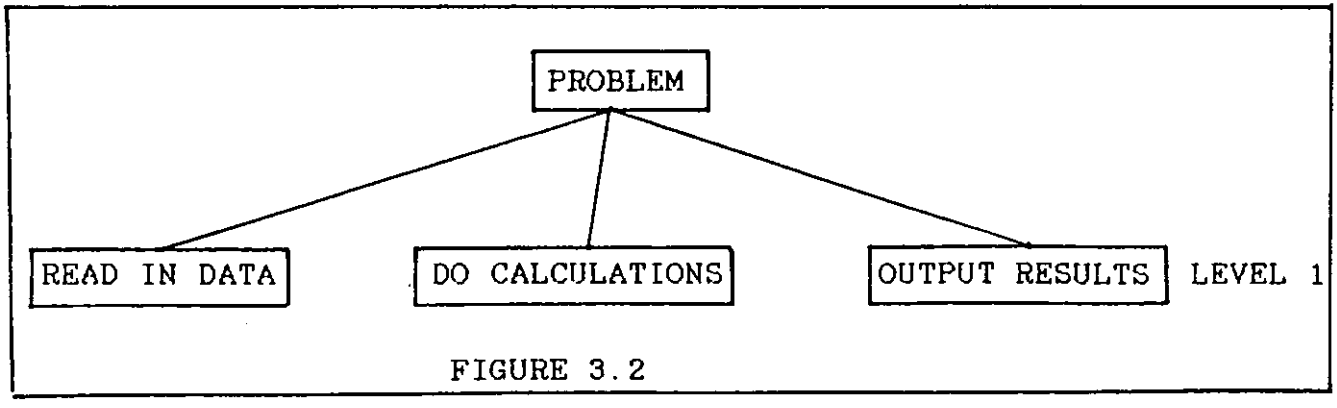


FIGURE 3.2

Typically, the upper levels contain statements of WHAT has to be done. These are then refined into statements of HOW it should be done. The advantages of using this method are:

1. The overall structure takes precedence over details and indeed should be complete before any details are considered.
2. The most important decisions are made first.
3. The programmer's mind is focussed initially on the problem rather than on the machine or the syntax of a programming language. The design then proceeds FROM the problem TO the machine.
4. Each level of the design provides a complete solution to the problem, which may be checked by another programmer (or by a machine), before proceeding to the next level.
5. The complexity is reduced. The programmer only has to consider one problem at a time.
6. If a design flaw is discovered the programmer may start again without having to scrap a lot of redundant code (or, worse still, patch up the faulty code).
7. Solutions developed in this manner may be coded into any programming language.

The basic building block of structure is the procedure or subroutine. A procedure is a section of a program that performs some particular task. Procedures should not be regarded as containers for chunks of code but rather as logical units, whose job it is to carry out some well-defined

function. It should be possible to state the purpose of each procedure in one short sentence. If this cannot be done, then the procedure should probably be refined further. The collective structure of the procedures in a program is of paramount importance and should be developed and tested before going on to consider the structure within each procedure.

If the scope of variable names can be limited to the procedures in which they occur, then the procedures may be written independently without fear of variable name clashes at a later stage in program development. Procedures written in this manner may also be used in many different programs.

3.2.3 CONTROL STRUCTURES

The controversy over the use of control structures is usually traced to a famous paper by Dijkstra in which he pointed out that unrestricted use of GOTO statements led to unnecessarily complex flow paths [37]. It had been proved as early as 1966, by Bohm and Jacopini [38], that any program could be written with only two control structures, one conditional and one iterative. In his subsequent work, Dijkstra has consistently advocated the use of simpler languages and has proposed a language that contains just two control structures [39].

It has been claimed that the discipline imposed by using only these basic structures improves the performance of even the best programmers [40]. There is also some psychological evidence to indicate that the choice of control structures does make a significant difference in programmer performance

[41]. This evidence supports the maxim that the number of bugs in any program is directly proportional to the square of the number of GOTO statements.

The essence of a good control structure is that it has one entry point and one exit point, with entry at the start of the structure and exit at the end. For iteration, the FOR/NEXT, REPEAT/UNTIL and WHILE/ENDWHILE structures meet these requirements. For selection, both the IF/THEN/ELSE and CASE/ENDCASE structures are generally considered to be satisfactory.

3.2.4 CODING STYLE

The use of a good coding style is generally regarded as an aid to making programs more readable. There is some evidence to suggest that the use of a good style has only a marginal influence on the comprehensibility of programs, but that most programmers regard it favourably and believe that it improves their performance [42]. Some important elements of coding style are:

1. Comments.

   Each procedure or unit should be prefaced by a brief description of its function. Low level comments which simply reiterate the operation of a particular statement are of little value. Comments should relate to the problem rather than to the program, so that "Find student with highest mark" is better than "Find maximum integer in array". In addition, long programs should contain a 'table of contents' at the beginning.

2. Identifiers

   Identifiers should be chosen carefully, to reflect the objects that they are intended to represent. Each procedure should contain a list of the variables used in it. In languages in which identifiers are limited to one or two characters each procedure should contain a dictionary of identifiers.

   e.g. VR = VAT RATE etc.

3. Text Format.

   Indentation is used to show where control structures begin and end. This helps the reader to see the logical structure of the program. It is especially effective when control structures are nested. The liberal use of spaces and blank lines is also helpful and it is generally considered good style to write just one statement per program line.

4. Brackets.

   It is possible in most languages to write long arithmetic and boolean statements without brackets, as there is a fixed order of precedence for both arithmetic and boolean operators. This, however, may cause difficulty for the reader and is not a good idea.

3.3 STRUCTURED PROGRAMMING IN EDUCATION

The overall goal of structured programming is to find simple solutions to difficult problems and to represent these solutions as clearly as possible. The key element in all of this is the idea of Top-Down design. Quite apart from the

world of computers, this is a skill which can be applied to a wide variety of problems. The ability to suppress the consideration of details until the overall structure has been completed is important in many activities and is essential in all complex activities.

It is likely that the majority of good problem-solvers have always worked in this way, but up to now the method has not been formally described. It is also likely that many good teachers have used such methods to demonstrate problem-solving activity for their classes but, so far, such methods have not been considered as objects of study in themselves and have not been explicitly taught in schools. Computer programming provides a unique opportunity for focussing on such problem-solving processes that is not afforded by other school subjects. When programming, it is possible to gain insight into one's own problem-solving strategy by examining that strategy as a program and then observing the outcomes of that strategy when the program is executed. By reflecting on the outcomes it is then possible to alter or improve components of the strategy until it is considered satisfactory.

An understanding of structured programming will also make students aware of the existence of design criteria such as clarity, efficiency and symmetry of structure. By designing solutions in a Top-Down manner, it is possible to foster a consciousness of the design decisions taken at each level. The student will be required to critically examine, and possibly reject, solutions which provide the correct results but which may violate some of the other criteria. In short,

students will become aware that getting the right answer is not the only goal in problem solving, but that there may be a variety of solutions to choose from, and that some solutions may be preferable to others.

There will be situations where students are unable to find a complete solution to a given problem. This may be frustrating but, if the Top-Down method is followed in these cases, the student will find a partial solution and clarify the problem. This should lead to the ability to know when to seek the teacher's help and to ask specific, well-directed questions.

Introducing students to structured programming is therefore a worthwhile educational goal because of its role in focussing on the problem-solving process and because of the insight it can provide into the whole area of computing. It is intended to teach students not only how to use the method, but also to make them fully aware of the method they are using and of its possible application in other areas of the curriculum.

CHAPTER 4

DESIGN OF A SUITABLE MINI-LANGUAGE

4.1 LANGUAGES FOR BEGINNERS' COURSES.

At present there seems to be no generally accepted educational philosophy guiding the design of programming courses at second level. The rationale for schools' programming courses may vary from vocational training for those who wish to pursue careers as programmers, to an extension of the mathematics syllabus. In many cases, however, the content is largely determined by the available hardware and much emphasis is placed on learning the syntax and semantics of whatever programming language is most easily implemented. While this is understandable in the present confused situation, it is not the best approach and is coming under increasing criticism. Moursound [43] has suggested that present second level courses in the United States may be doing more harm than good and that students who have taken such courses may be at a disadvantage if they proceed to take a University Computer Science course.

It is often argued that the use of BASIC in second level courses is at the root of the problem. While it is recognised that BASIC is structurally deficient and to be avoided whenever possible, the same problems can arise with any programming language if the primary aim is to teach the programming language rather than to teach programming. It is vital, particularly for novices, to separate the task of

solving problems from the task of learning the rules and regulations of a programming language. The programming language (i.e. the implementation language) should be seen as a tool for communicating the completed solution to the computer, rather than as a means of finding solutions in the first instance, and the emphasis should be firmly placed on finding well structured, elegant solutions. In the school situation, programs are usually written, tested and then never used again. It is easy to overlook the importance of structure under these circumstances, but to do so is to miss an important educational opportunity.

While the methods of structured programming were developed to allow programmers to deal with large complex problems, the types of problems used in a first programming course will not be very large or very complex. This should not be used as an argument for postponing the teaching of structured programming until some later stage, as these methods are just as powerful when applied to small problems and are likely to be transferable to many other situations. When beginning a first course in programming, students are usually eager and well motivated, so that the first concepts absorbed by them are likely to become well established and will be difficult to change at a later stage. Therefore, it is important to design courses that help students to think in an orderly and precise manner right from the start. It is also important to find problems which illustrate and reinforce important programming concepts and which are of interest to, and within the reach of, our students. The temptation to choose problems which

merely demonstrate the capabilities of the particular machine being used should be resisted.

4.2 THE NEED FOR A MINI-LANGUAGE

In designing a suitable problem-solving language, it is important to be aware that language is not only a medium of expression but also an instrument of reason. Bruner has pointed out that language is "not only the medium of exchange but the instrument that the learner can then use himself in bringing order to the environment" [44]. Any language or notation used should have the important characteristic of relieving the learner of unnecessary work, thus allowing concentration on the problem at hand. This can be achieved by designing a language which:

1. Contains the minimum number of different concepts with clear and simple rules for their combination.
2. Is easy to learn and to use.
3. Is applicable to a wide range of problems.
4. Has consistent application of the same rules in the same way throughout.
5. Contains suitable control structures which are available, or may be simulated, in most programming (implementation) languages.

In short, the language should provide a good conceptual framework for thinking about algorithms and should be easily translated into any programming language with the minimum of translation rules.

If a course is to be designed which attempts to emphasise the

essentials of programming rather than the details of some particular programming language, then a mini-language consisting of just the vital statements should be used. Most of the widely available programming languages contain far more constructs than are necessary, and there is evidence to suggest that even professional programmers use only a small subset of their particular language a large proportion of the time [45]. The two most important concepts in programming are iteration and selection, and the choice of control structures for these must be made with care. There are numerous control structures available for both selection and iteration but learning all of them confers no advantage on the student. Indeed it may hinder the development of the underlying concepts due to the inevitable concentration on the syntax and semantics of the large number of structures, and the requirement to remember the appropriate circumstances in which to use each individual structure. Another danger associated with this approach is that the student may be convinced that there is nothing more to programming than knowing all about the various control structures.

Therefore, the approach adopted was to single out one construct for iteration and one for selection. In each case, the most general construct was chosen so that it would work in all situations, thus relieving the student of the need to remember which construct to use under which set of circumstances. Using this approach, there was also less syntax to be learnt and this reduced the amount of specific, low level knowledge to be absorbed, allowing for concentration

on the underlying concepts. This restriction also meant that fewer translation rules were required to convert the solutions into implementable programming languages, which was also an important consideration.

This idea of using a mini-language for introductory courses had been tested with undergraduates by Riley [46] and Campbell [47]. In both cases, a subset of PASCAL was used and problem-solving techniques were emphasised. In each case, a significantly positive effect was noted on subsequent performance in traditional PASCAL courses. Campbell reported an improvement of an entire grade on average.

4.3. MINI-LANGUAGE DEFINITION

The mini-language which was designed contained very few syntax rules and, equally important, very few translation rules for converting solutions into programming languages. It was hoped that this restriction would encourage students to approach similar problems in similar ways, and so help them to develop a consistent style of programming.

4.3.1 ITERATIVE STRUCTURES

The iterative structure used was WHILE/ENDWHILE as it is the most general of those available. All FOR/NEXT and REPEAT/UNTIL loops may be written as WHILE/ENDWHILE loops, but there are WHILE/ENDWHILE loops which may not be written in either of the other forms. The FOR/NEXT loop in figure 4.1 is of the most general type possible and yet is very simply 'matched' by the WHILE/ENDWHILE loop on the right.

```
FOR COUNT:= N TO M STEP L                COUNT:= N
      .                                  WHILE COUNT <= M DO
      .                                     .
      .                                     .
      .                                     .
      .                                     COUNT:= COUNT + L
NEXT COUNT                               ENDWHILE
```

FIGURE 4.1

Another objection to the FOR/NEXT loop is the number of variations in the way it can be written as N, M and L may be variables or constants and may be positive or negative. The worst aspect of all, however, is its lack of clarity because:

1. It is not clear what value COUNT has after the loop has been executed.
2. It is not clear what happens if N = M, or if N > M and L is positive.
3. It is not clear what happens if the values of COUNT, N, M and L are changed within the body of the loop (and most implementations allow this to be done!).

These problems do not arise with the WHILE/ENDWHILE loop. The loop control variable COUNT is explicitly incremented and so there can be no doubt as to its value at any time before, during or after the iteration. Likewise, the boolean condition at the entry point to the loop is perfectly explicit and so there is no doubt as to when the loop may or may not be executed. This superiority of the WHILE construct is well recognised and Metzler has reported an attempt to teach the WHILE construct before the FOR/NEXT in BASIC, despite the fact that the WHILE construct has to be simulated using IF and GOTO statements [48].

Every REPEAT/UNTIL loop can be written as a WHILE loop by reversing its boolean condition (Figure 4.2). The opposite is not the case, as every REPEAT/UNTIL loop is executed at least once because there is no guard on entry to the structure. There are numerous situations in which a loop should not be executed at all, and this cannot be achieved using a REPEAT/UNTIL loop.

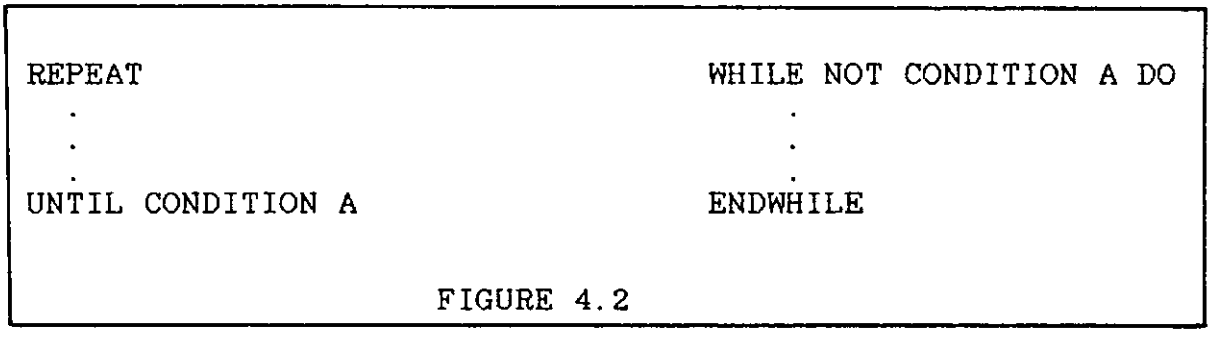


FIGURE 4.2

Although it is possible to write all REPEAT/UNTIL loops as WHILE/ENDWHILE loops, this may not always be desirable from the point of view of simplicity and clarity. In particular, there are occasions when some variable in the boolean CONDITION A is initialised within the body of the loop. In this case the NOT CONDITION A expression above would be invalid because it contained an undefined variable. This difficulty may be overcome by introducing another variable (often a boolean variable) in the WHILE condition but it is probably better to use a REPEAT/UNTIL loop instead. These circumstances are rare and so this slight difficulty should not be allowed to interfere with the language design.

The big advantage of the WHILE/ENDWHILE structure is that the condition is tested before the loop is entered. This is generally regarded as better programming practice [45].

4.3.2 DECISION STRUCTURES

The normal practice with decision structures is to use the IF/THEN/ELSE structure for two-way decisions and CASE/ENDCASE for multiway decisions. However, the CASE structure was rejected because any list of constants in a CASE statement may be written as a boolean expression, but the converse is not true. Therefore the IF statement, which utilises boolean conditions rather than lists of constants is more general:

i.e. CASE X$ OF

a,b,c

is equivalent to

IF X$ = "a" OR X$ = "b" OR X$ = "c"

Conversely, even simple boolean expressions such as:

X > 0 AND X < 10 (where X is of type REAL)

may not be expressed as a list of constants. The remaining choice, therefore, was between the various kinds of IF structures.

The most general IF structure, IF/THEN/ELIF/ELSE, was chosen as this supports one-way, two-way and multiway decisions. (ELIF is equivalent to ELSE IF). The use of ELIF enables students to avoid deeply nested structures when they are not necessary although, of course, nesting of structures is still possible. The ELIF structure in figure 4.3 is a much simpler and clearer representation than the nested structure. The conditions "light is green", "light is orange" and ELSE (i.e. "light is red") are all at the same semantic level and so should be represented as such in the program text. This is not possible using the IF/THEN/ELSE structure.

Furthermore, it is possible to explicitly state all the conditions (i.e. avoid the use of ELSE) using the ELIF structure and this is often desirable.

```
IF light is green THEN
    Go
ELIF light is orange THEN
    Stop if you can
ELSE
    Stop
ENDIF
```

```
IF light is green THEN
    Go
ELSE
    IF light is orange THEN
        Stop if you can
    ELSE
        Stop
    ENDIF
ENDIF
```

FIGURE 4.3

A major advantage of choosing these two control structures is that they are very similar. In both cases entry to the structure (and to any branch in the case of the IF structure) is controlled by boolean expressions. This unity is more satisfactory than a mixture of loop counters, lists of constants and boolean expressions, which would be the case if all the usual structures were introduced. This, of course, also implies that boolean expressions are extremely important and must be dealt with extensively and carefully in the course.

4.3.3 OTHER STATEMENTS AND STRUCTURES

The other statements and structures included in the mini-language were:

1. An INPUT statement.
2. A PRINT statement.
3. An assignment statement.
4. Variables of type REAL and STRING.

5. READ and DATA statements.
6. Procedures.
7. Arrays.

4.4 DIAGRAMMATIC REPRESENTATION OF PROGRAM DESIGN

It was necessary to devise a system of representing the hierarchical nature of good Top-Down designs in a manner which could be easily translated into programming languages. This system should not alone represent the hierarchy of the solution but should assist the learner to discover it. It should also, of course, be capable of supporting the chosen control structures. There are three principal systems used to represent algorithms in the popular textbooks. These are a) flowcharts which are mainly associated with programming in BASIC, b) pseudocode which is mainly used in PASCAL textbooks and c) structure diagrams which are not as widespread and are often associated with COMAL.

4.4.1 FLOWCHARTS

Flowcharts may be criticised on the grounds that:

1. They graphically depict ANY flow of control and discourage the discipline necessary to maintain good structure.
2. They force the designer to concentrate on the most detailed aspects of the problem before the overall design is complete.
3. The flowchart and the program code are at the same semantic level and so the flowchart confers no advantage on the reader.
4. Levels of detail are very easily mixed and confused.

5. They have no provision for representing multiway decisions.
6. There is no symbol to represent a loop.

As flowcharts are organised around conditional and unconditional jumps, it is extremely difficult to see how a set of rules could be drawn up to translate them into a language which does not contain a GOTO statement. Flowcharts, then, distract attention from the important functional relationships in the overall design. They highlight the flow of control at the expense of inherent structure and so are of no value at all in helping to develop an awareness of the importance of structure.

4.4.2 PSEUDOCODE

There are many varieties of pseudocode but most of them are a compromise between English and PASCAL. This is a popular and useful method of program development and does attempt to represent the overall hierarchical nature of the solution. It is usual to begin with a very high level description of the solution as a list of indexed points. Each point is then refined, if necessary, into a further list of indexed sub-points and this process is continued until the program is completely broken down into a set of simple sub-tasks. This is the classic Top-Down approach but, unfortunately, may lead to a solution spread over a number of pages of text. In addition, there is no obvious representation (apart from the indices) of the relationships between the various sub-tasks. It is also an extremely 'wordy' method and is not easily represented on a blackboard or overhead projector.

4.4.3 STRUCTURE DIAGRAMS.

Structure diagrams are developed in exactly the same way as pseudocode but the finished product represents both the hierarchy and the relationships between the parts in a much clearer manner. They are very suitable for use in schools because:

1. They force students to use a Top-Down design.
2. They show overall structure at a glance.
3. They are hierarchical and support the refinement process.
4. They are very different from any programming language and so reinforce the idea of program design as a separate task from coding.
5. They embody very few syntax rules of their own and so the technique of constructing them can be learnt quickly and easily.
6. It is easy to modify the lower levels of the diagram, without disturbing the upper levels. This is important as problem-solving is often a trial and error process. This flexibility should encourage students to examine their own designs critically and continuously.
7. At every level there is a complete solution. This may help students to feel that they are making progress and encourage them to persevere.
8. Students may have any level of design checked by the teacher before proceeding to the next level.
9. The teacher may supply the upper levels and require students to further refine the solution.

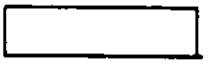
10. They may be translated directly into any computer language, at the keyboard, by following a few simple rules.
11. They are very suitable for use on both blackboards and overhead projectors.

A form of structure diagram has been used by Kelly [49] and Atherton [50] but this does not adequately represent the mini-language outlined above. A variation of these diagrams was therefore designed to match the structures of the proposed mini-language. A properly drawn diagram of this type is a good reflection of one's problem analysis. It also represents the exact ordering of the statements within the program and so may be readily translated into a programming language.

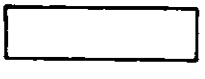
Each diagram is a tree-like structure (the tree is upside down) and may consist of:

(i) A ROOT box: 

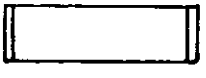
This contains the title of the program.

(ii) NODE boxes: 

These are boxes that have been further refined. They contain headings which may be used as REM statements.

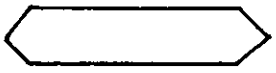
(iii) LEAF boxes: 

These are boxes that have not been further refined. They contain actions.

(iv) PROCEDURE boxes: 

These contain procedure names.

(v) THE IF SYMBOL: ●

This denotes the beginning and the end of an alternative control structure. Beneath the IF symbol, and attached to it, there must be at least one DECISION BOX: 

Each decision box contains a boolean condition guarding entry to a block of the IF structure. The last box may contain the word ELSE denoting an ELSE clause. If all of the previous conditions have been found to be FALSE, then the ELSE branch is executed by default.

(vi) LOOP CONTROL boxes: 

These denote the beginning and the end of a loop. Each one contains a boolean expression which 'guards' entry to a loop. The boxes below it, and attached to it, contain the statements of the body of the loop which are executed repeatedly as long as the boolean expression is TRUE.

4.5. TRANSLATION OF DIAGRAMS INTO PROGRAMMING LANGUAGES

It was necessary to specify exactly how the diagrammatic solutions should be translated into programming languages, as otherwise the diagrams would be useless. This point is rather neglected in most textbooks that use either structure diagrams or flowcharts; possibly because of the near impossibility of describing the translation process if all the various looping and branching structures are used. Nevertheless, it is vital that this is done, as otherwise the student is burdened with

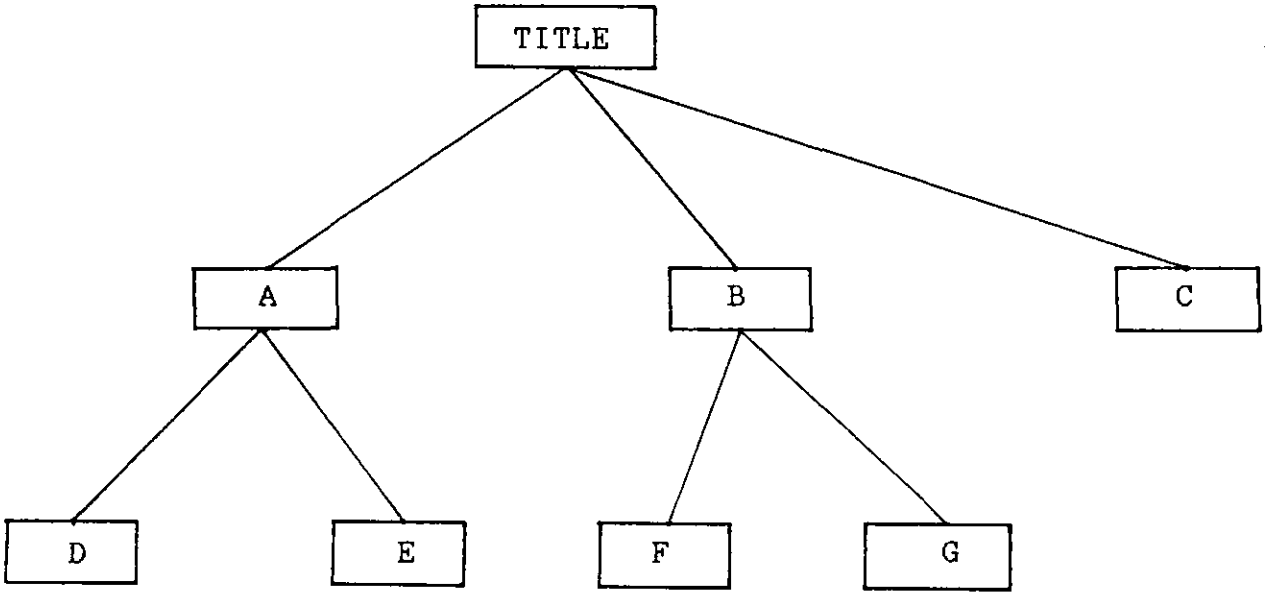
unnecessary effort which may distract from the problem at hand. Ideally, the translation rules should be simple, unambiguous and as few as possible, making the translation process a purely 'mechanical' one. Once the translation rules are learned, the student should be able to write programs at the keyboard directly from a diagram in whatever implementation language is being used. This part of the programming process should be viewed by the student as fairly trivial and not nearly as important as designing the diagram in the first place.

4.5.1 TRANSLATION OF SEQUENTIAL STATEMENTS

The general method for translating any diagram into a programming language is as follows:

1. The diagram is a tree (upside down).
2. The tree consists of a root, leaves and nodes. The root is the box containing the name of the problem. Nodes are boxes that have been further refined. Leaves are boxes that have not been refined.
3. Starting at the root 'walk' around the tree keeping an imaginary left hand on it at all times. Whenever a leaf is encountered, the appropriate statement is written in the programming language. Nothing is written at a node.

An example of a simple diagram and the program derived from it is shown in figure 4.4. This general scheme is the same for all languages. Differences only arise when conditional and iterative control structures need to be translated.



Program derived from diagram :

```
D
E
F
G
C
```

Note: A and B have both been further refined and therefore are not statements in the program. They may however be used as REM statements because A describes the process carried out by executing D and E, and B describes the process carried out by executing F and G.

FIGURE 4.4

4.5.2. TRANSLATION OF THE CONDITIONAL CONTROL STRUCTURE

The start and end of a conditional structure is represented in the diagrams by the symbol ● . The example in figure 4.5 is interpreted as follows:

1. Once the structure is entered the first condition on the left is tested.
2. a) If this is found to be TRUE then the statements below it (guarded by it) are executed. The other conditions in the structure are not then tested as only one branch of the structure may be executed.

   b) If it is found to be FALSE then the next condition to the right is tested.
3. This process is repeated until either one condition is found to be TRUE or all the conditions have been found to be FALSE.

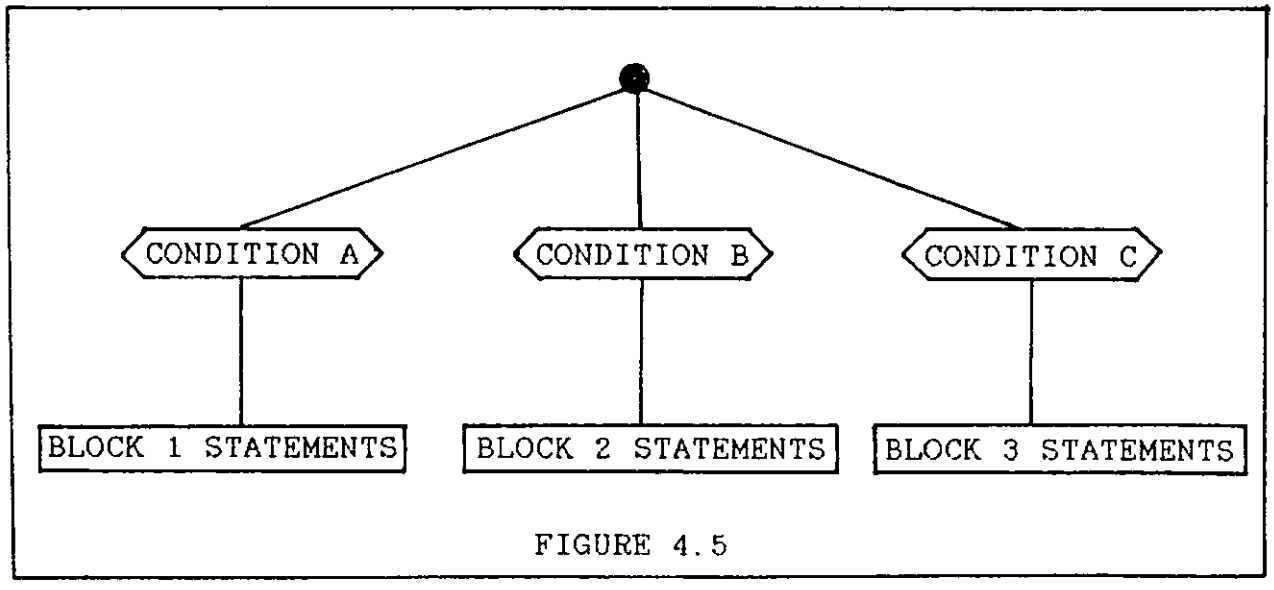


FIGURE 4.5

It may be noted that, at most, one branch is executed but that it is also possible that no branch is executed if all the

conditions are found to be FALSE. There is no upper limit on the number of branches but, of course, there must be at least one.

The diagram in figure 4.5 may be translated into the various languages as shown in figure 4.6. The COMAL editor automatically indents control structures as shown but in PASCAL the indentation must be added by the programmer. With regard to the Applesoft BASIC program it may be noted that:

1. The 'reversal' of the boolean conditions can be achieved quite readily by using the NOT operator or (preferably) by using de Morgan's law.
2. It is possible to derive a simpler translation rule, which doesn't require reversal of the conditions, by using multi-statement lines. This works quite well if there are just a few statements in each block, but fails if there are more statements in a block than will fit on a program line. It is also very awkward to use this method when nested IFs are involved.
3. The editor does not allow statements to be indented as shown, and actually removes indentation if included by the programmer.

The style of coding used in the BASIC examples is quite unlike the norm found in textbooks. The advantages of this style are that control structures are clearly indicated with just one entry point and one exit point, GOTOs are confined to jumps within a control structure and the statements governed by any

METANIC COMAL

```
100 IF CONDITION A THEN
110    BLOCK 1 STATEMENTS
120 ELIF CONDITION B THEN
130    BLOCK 2 STATEMENTS
140 ELIF CONDITION C THEN
150    BLOCK 3 STATEMENTS
160 ENDIF
```

APPLE PASCAL

```
IF CONDITION A THEN
   BEGIN
   BLOCK 1 STATEMENTS;
   END
ELSE IF CONDITION B THEN
   BEGIN
   BLOCK 2 STATEMENTS;
   END
ELSE IF CONDITION C THEN
   BEGIN
   BLOCK 3 STATEMENTS;
   END;
END.
```

APPLESOFT BASIC

```
100 REM STARTIF
110 IF NOT (CONDITION A) THEN GOTO 140
120    BLOCK 1 STATEMENTS
130    GOTO 200
140 IF NOT (CONDITION B) THEN GOTO 170
150    BLOCK 2 STATEMENTS
160    GOTO 200
170 IF NOT (CONDITION C) THEN GOTO 200
190    BLOCK 3 STATEMENTS
200 REM ENDIF
```

B.B.C. BASIC

```
100 REM STARTIF
110 IF CONDITION A THEN 120 ELSE 140
120    BLOCK 1 STATEMENTS
130    GOTO 190
140 IF CONDITION B THEN 150 ELSE 170
150    BLOCK 2 STATEMENTS
160    GOTO 190
170 IF CONDITION C THEN 180 ELSE 190
180    BLOCK 3 STATEMENTS
190 REM ENDIF
```

FIGURE 4.6

particular condition are written immediately after that condition.

The final conditional box may contain the word ELSE, rather than a boolean condition. In this case the last branch of the structure is executed if all the previous conditions have been found to be FALSE. If there is an ELSE branch in the structure, as in figure 4.7, it may be coded into the various languages as in figure 4.8 (see page 57).

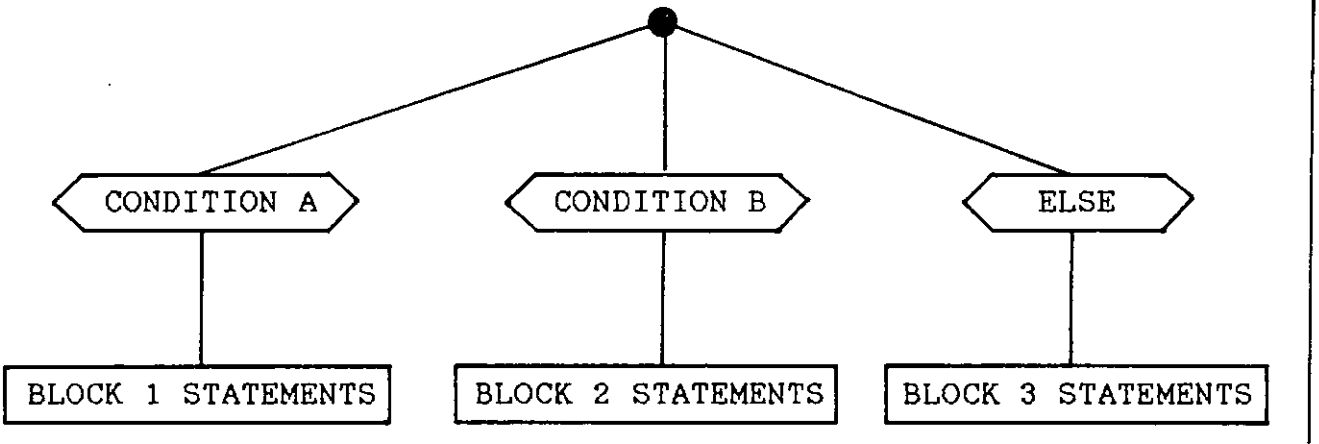


FIGURE 4.7

4.5.3 TRANSLATION OF THE ITERATIVE CONTROL STRUCTURE

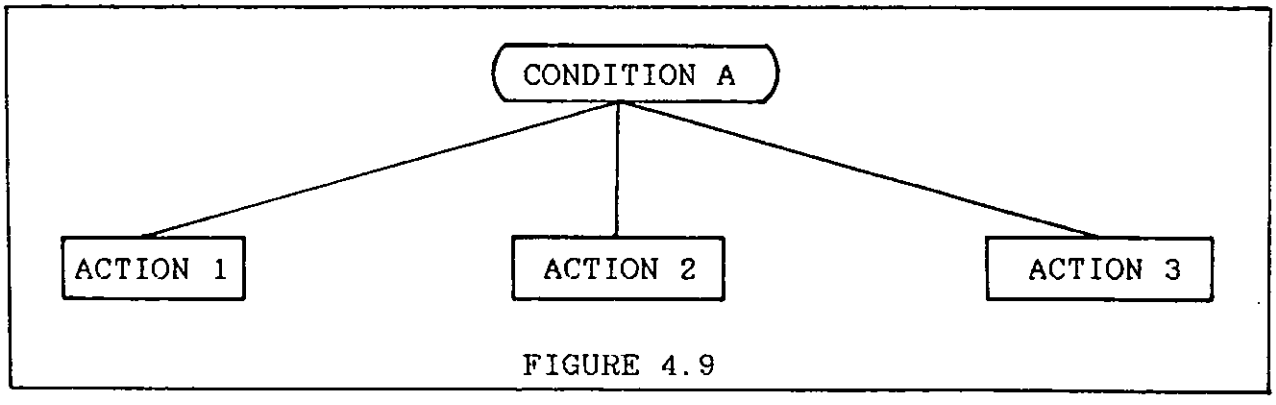


FIGURE 4.9

The diagram in figure 4.9 is interpreted as follows:

1. The condition in the round box is tested.

METANIC COMAL

```
100 IF CONDITION A THEN
110    BLOCK 1 STATEMENTS
120 ELIF CONDITION B THEN
130    BLOCK 2 STATEMENTS
140 ELSE
150    BLOCK 3 STATEMENTS
160 ENDIF
```

APPLE PASCAL

```
IF CONDITION A THEN
   BEGIN
   BLOCK 1 STATEMENTS;
   END
ELSE IF CONDITION B THEN
   BEGIN
   BLOCK 2 STATEMENTS;
   END
ELSE
   BEGIN
   BLOCK 3 STATEMENTS;
   END;
END.
```

APPLESOFT BASIC

```
100 REM STARTIF
110 IF NOT (CONDITION A) THEN GOTO 140
120    BLOCK 1 STATEMENTS
130    GOTO 200
140 IF NOT (CONDITION B) THEN GOTO 170
150    BLOCK 2 STATEMENTS
160    GOTO 200
170 REM ELSE BRANCH
180    BLOCK 3 STATEMENTS
200 REM ENDIF
```

B.B.C. BASIC

```
100 REM STARTIF
110 IF CONDITION A THEN 120 ELSE 140
120    BLOCK 1 STATEMENTS
130    GOTO 190
140 IF CONDITION B THEN 150 ELSE 170
150    BLOCK 2 STATEMENTS
160    GOTO 190
170 REM ELSE BRANCH
180    BLOCK 3 STATEMENTS
190 REM ENDIF
```

FIGURE 4.8

2. If this is found to be TRUE then the statements below it (guarded by it) are executed. The condition is then tested again.
3. This process is continued until the condition is found to be FALSE.

Note: If the condition is found to be FALSE on the first test then the guarded statements are not executed at all. The diagram in figure 4.9 may be translated into the various languages as shown in figure 4.10.

4.6 CHOICE OF IMPLEMENTATION LANGUAGE

Although one of the chief aims of the course was to teach programming concepts independently of any particular programming language, it was still necessary to choose a computer language for implementing and testing programs. The languages available were APPLESOFT BASIC, APPLE PASCAL, METANIC COMAL (for Apple machines) and B.B.C BASIC.

It is clear from the coded examples that both COMAL and PASCAL support the chosen structures in a very clear manner. With both of the BASICs the structures have to be simulated. The conditional statement is particularly tedious to construct in BASIC. This is not due to any fault in the chosen control structure but because both BASICs are structurally deficient languages. Applesoft BASIC is particularly bad as it does not support any control structures, apart from the IF/THEN conditional statement (with no ELSE branch) and the FOR/NEXT loop. Neither does it support procedures. It also has an appallingly bad editor, which is another important consideration. B.B.C. BASIC is slightly more structured and has

METANIC COMAL

```
100 WHILE CONDITION A DO
110    ACTION 1
120    ACTION 2
130    ACTION 3
140 ENDWHILE
```

APPLE PASCAL

```
WHILE CONDITION A DO
   BEGIN
   ACTION 1;
   ACTION 2;
   ACTION 3;
   END.
```

APPLESOFT BASIC

```
100 REM STARTLOOP
110 IF NOT (CONDITION A) THEN GOTO 160
120    ACTION 1
130    ACTION 2
140    ACTION 3
150    GOTO 100
160 REM ENDLOOP
```

B.B.C. BASIC

```
100 REM STARTLOOP
110 IF CONDITION A THEN 120 ELSE 160
120    ACTION 1
130    ACTION 2
140    ACTION 3
150    GOTO 100
160 REM ENDLOOP
```

FIGURE 4.10

a good editor. It supports procedures, the REPEAT/UNTIL loop and the IF/THEN/ELSE conditional statement. This latter structure is not much of an improvement on the Applesoft IF/THEN statement as the complete statement must be written on one program line.

The choice of language then was between METANIC COMAL and APPLE PASCAL. This was an easy decision as the operating environment of COMAL is simpler. It was feared that learning how to manipulate the PASCAL editor, filer, compiler etc. would take up most of the available time, leaving no time for the real purpose of the course, i.e. solving problems. It was also felt that an interpreted language, rather than a compiled one, was better for beginners. This was because their initial efforts were sure to contain many errors and the interpreter could give them immediate feedback. COMAL is particularly good in this respect as it checks the syntax of each line as it is typed and reports any errors found. It also performs a pre-RUN check on each program to ensure that all control structures are properly nested and closed. If any error of this nature is found then it is reported by the system. In effect, this means that any program which can be RUN is free of syntax errors and so if the desired result is not achieved it can be assumed that there is a logical error. This knowledge, combined with the system-forced indentation, is helpful when debugging programs. A further reason for choosing COMAL was that it allows external library procedures to be incorporated into programs in a very simple manner.

CHAPTER 5

REVIEW OF THE AVAILABLE TEXTBOOKS

5.1 INTRODUCTION

A review of the currently available textbooks for second level courses was carried out before developing the course materials. The purposes of this review were:

1. To see if such textbooks might be suitable for the course.
2. To gain some insight into what are the accepted norms for such courses.
3. To show how the mini-language and the structure diagrams can be used to solve the problems that are used in these books and to compare the solutions developed in this way with those in the books.
4. To find problems that might be used in the course.

Five books were reviewed [49,50,51,52,7]. Two of these [49,51] are the current best-sellers in Irish second level schools. The other two COMAL books [50,52] were reviewed briefly because COMAL was the chosen implementation language. The final book [7] was reviewed because it has been distributed to all second level schools by the Department of Education and is concerned with Top-Down programming, structure diagrams and COMAL.

5.2 BASIC COMPUTER PROGRAMMING FOR STUDENTS [51]

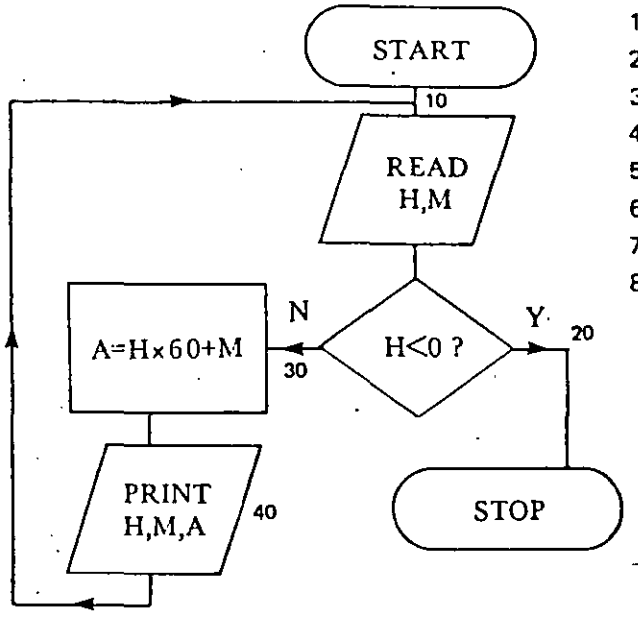
In the preface of this book the authors state that it was "produced with the object of enabling students of all age groups to communicate with computers using what is often regarded as the easiest and quickest computer language to learn - BASIC". In chapter 1 (p.9) they state that "the process of making an algorithm acceptable to a computer is called programming". These two statements would suggest that the emphasis of the book is on coding in BASIC and this is indeed the case. In the early chapters there is no clear distinction drawn between problem-solving, coding and editing. Most of the examples given are designed to show how BASIC statements such as READ, LET and PRINT work, rather than to solve any stated problem.

5.2.1 FLOWCHARTS

Initially, flowcharts are used to develop programs but after page 100 they are discarded. In the rest of the book, if an algorithm is developed before coding, it is done by writing a sentence describing each section of the program. In most cases, the finished program is presented without any preliminary discussion of the algorithm. Figure 5.1 (from p.51) is a typical example of the way in which flowcharts are used in the book. The flowchart is an exact, statement by statement, replica of the code and so is of little value. Even in such a simple problem, the structure diagram allows the programmer to seek a solution in a Top-Down manner. The completed diagram (figure 5.2) also tells us much more about how the problem was approached.

Create a program which reads a number of hours and minutes from data as a pair of values, calculates and prints the total number of minutes. e.g. 5,2 = 5 hours and 2 minutes. The *dummy value* used is minus hours (-H).

Solution




```
10 READ H,M
20 IF H<0 THEN 80
30 LET A=H*60 + M
40 PRINT H,M,"=",A
50 GOTO 10
60 DATA 5,2,3,45,15,16,32,4,24,0
70 DATA 48,0,-5,0
80 END
```

FIGURE 5.1

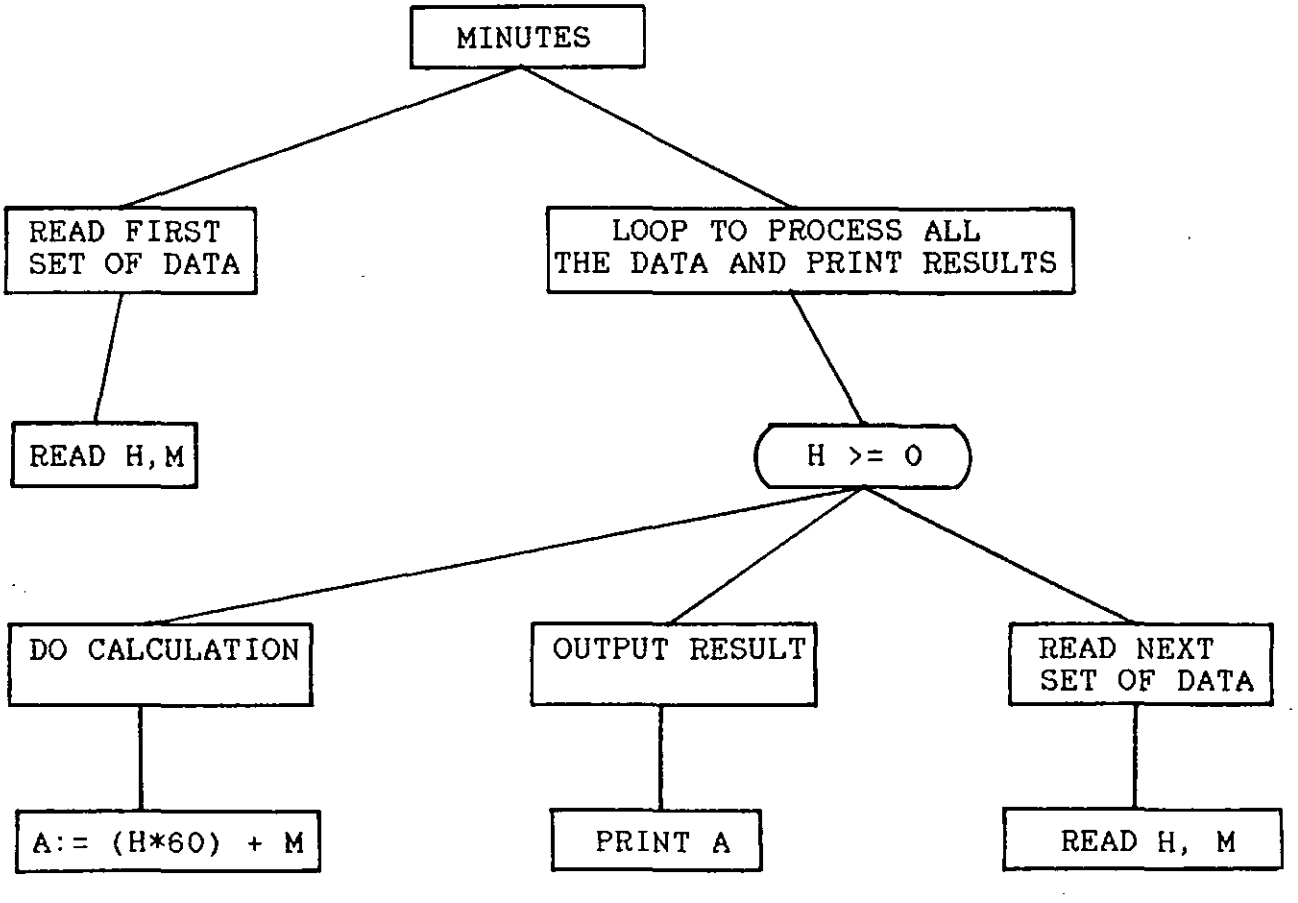


FIGURE 5.2.

The problem in figure 5.3 (from p.53) is re-solved using the mini-language and a structure diagram in figure 5.4 . The structure diagram illustrates the overall approach to the problem in a clearer fashion and is much easier to understand. The code developed from the flowchart involves four jumps and does not indicate where the loop begins and ends. This may be contrasted with the BASIC program derived from the structure diagram:

```
10  READ A
20  REM STARTLOOP
30    IF A < 0 THEN 70
40    IF A >= 40 AND A <= 50 THEN PRINT A
50    READ A
60    GOTO 20
70  REM ENDLOOP
```

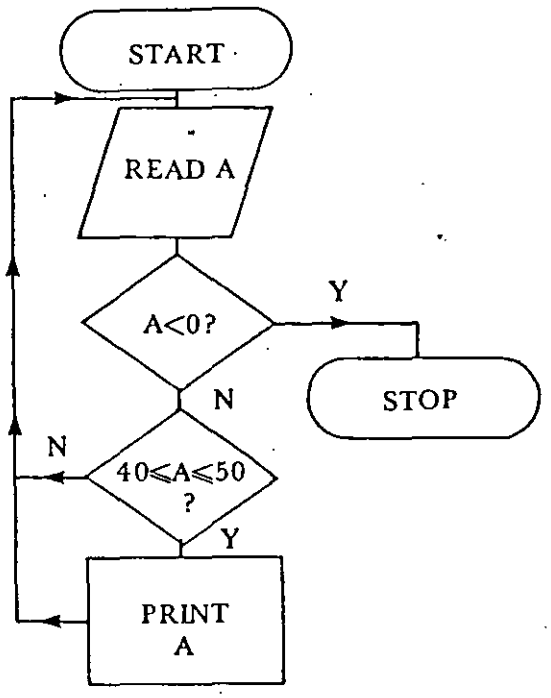
In figure 5.5 (from p.90) it is impossible to understand how the solution works without examining the flowchart in minute detail. This, in turn, indicates that in developing the solution it was necessary to consider all the minor details first and then build them up to arrive at the overall solution. This approach confers no advantage over coding the program directly in BASIC. This is in marked contrast to the structure diagram (figure 5.6) where it is obvious from the first few lines that the program is accepting positive numbers from the keyboard, searching the data for each one and terminating when a negative number is input.

5.2.2 SELECTION AND ITERATION

As the stated purpose of the book is to teach BASIC, there is little attention paid to the underlying programming structures. There is no clear distinction drawn between looping and branching. This is because both are introduced together in chapter 4 for the purpose of showing how the GOTO statement

Design a program to read an unknown quantity of values in the range 0 to 100 and to print those in the range 40 to 50 inclusive. A negative value is to be used to terminate the data.

Solution




```
10 READ A
15 DATA 47,8,42,75,41,39,95,-2
20 IF A<0 THEN 60
25 IF A<40 THEN 50
30 IF A>50 THEN 50
40 PRINT A
50 GOTO 10
60 END
```

FIGURE 5.3

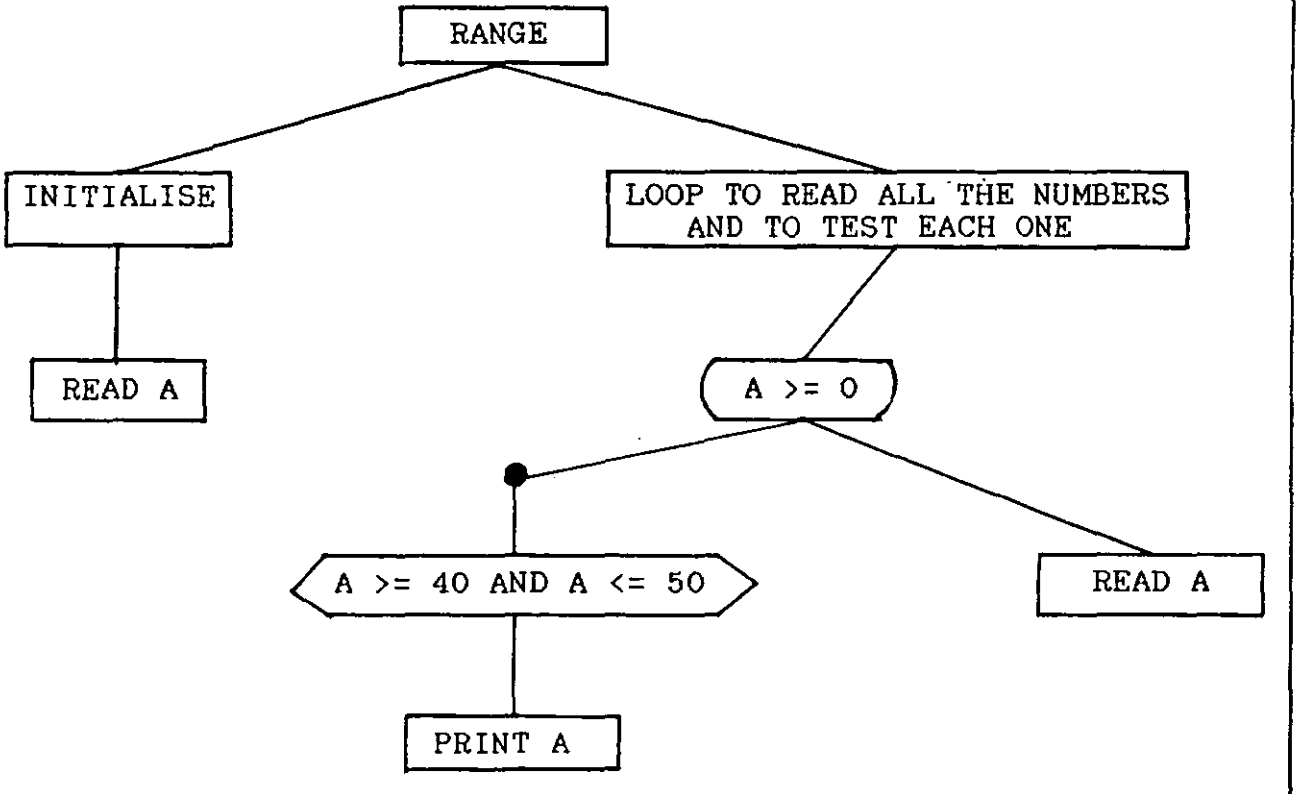
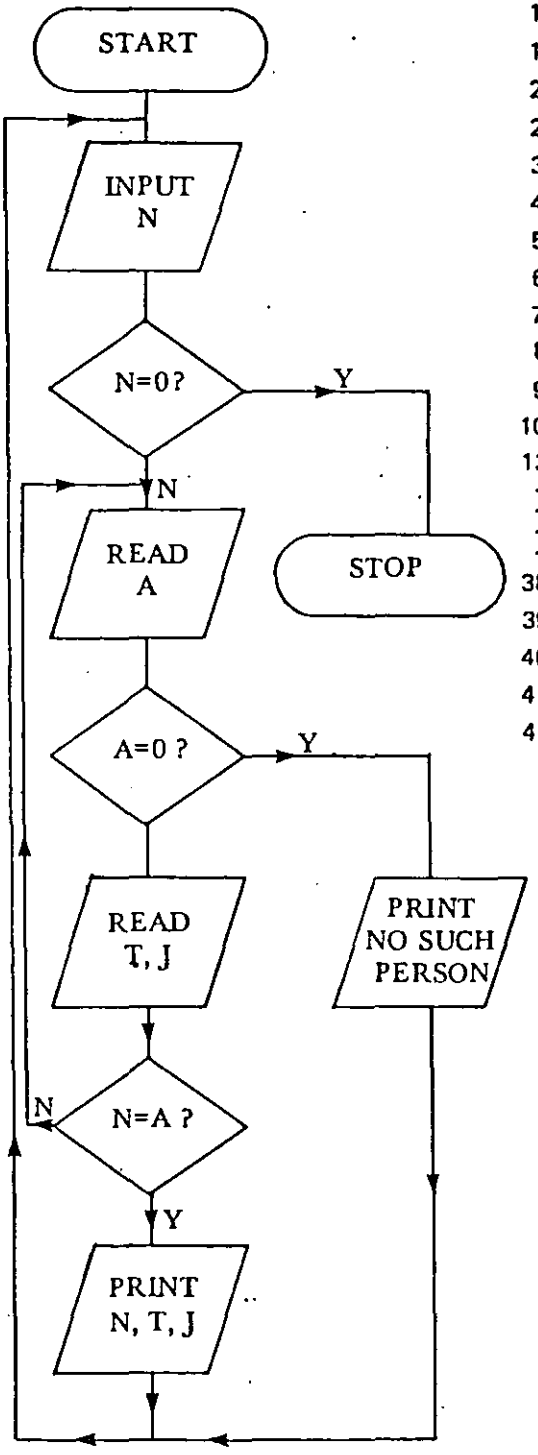


FIGURE 5.4

A set of data comprising a staff number, name and job description is provided for each of a number of employees. Create a program which finds the job description of an employee whose staff number is entered from a terminal.

Generalize this program to repeat the process until the value 0 is input for the staff number.

Flowchart



Program

```
 10 PRINT "ENTER EMPLOYEE NUMBER"
 15 PRINT "ENTER 0 TO END"
 20 INPUT N
 25 IF N=0 THEN 420
 30 READ A
 40 IF A=0 THEN 100
 50 READ T$,J$
 60 IF N=A THEN 80
 70 GOTO 30
 80 PRINT N;T$;" HAS JOB ";J$
 90 GOTO 400
100 PRINT "NO SUCH EMPLOYEE"
130 DATA 16,"JOE SMITH","CLERK"
    .
    .
    .
380 DATA 768,"ALAN HUGHES","DRIVER"
390 DATA 0
400 RESTORE
410 GOTO 10
420 END
```

FIGURE 5.5

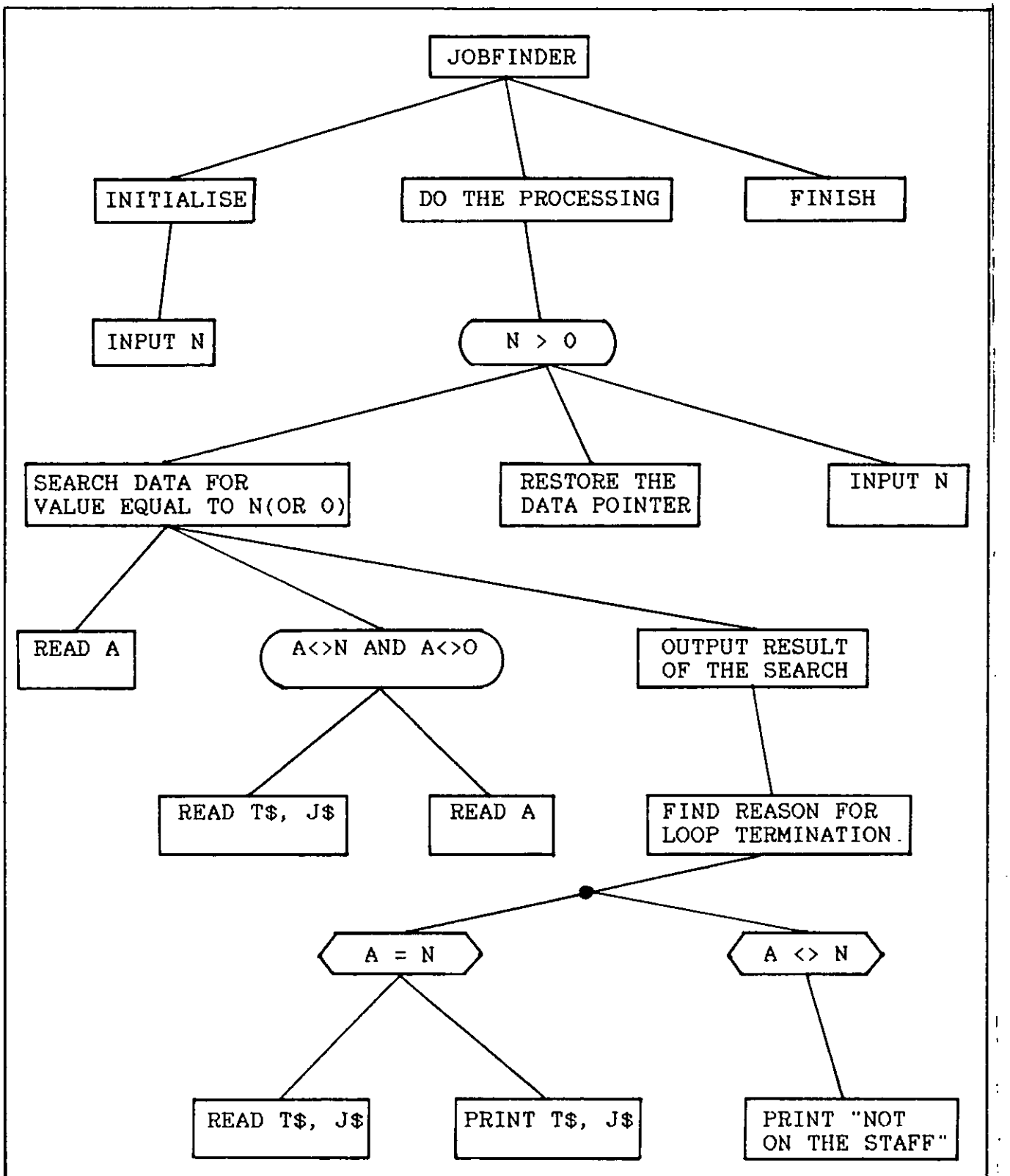

JOBFINDER
INITIALISE
DO THE PROCESSING
FINISH
INPUT N
N > 0
SEARCH DATA FOR VALUE EQUAL TO N(OR 0)
RESTORE THE DATA POINTER
INPUT N
READ A
A<>N AND A<>O
OUTPUT RESULT OF THE SEARCH
READ T$, J$
READ A
FIND REASON FOR LOOP TERMINATION
A = N
A <> N
READ T$, J$
PRINT T$, J$
PRINT "NOT ON THE STAFF"

FIGURE 5.6

works. It is felt that this approach is unwise as looping and branching are vital concepts and the GOTO statement exists merely to facilitate coding of loops and branches in BASIC.

The first program in the book that contains a loop (from p.46) is as follows:

```
10  READ H, R
20  LET P=H*R
30  PRINT P
40  GOTO 10
50  DATA 6,1.20, 7, 1.40, 5, 0.80
60  DATA 4, 1.50
70  END
```

This program contains an infinite loop and cannot terminate properly. The authors discuss the 'workings' of the program in detail and indicate that it will eventually terminate with an 'out of data' error. This is bad programming practice and an unsatisfactory way to introduce the important topic of loop construction. The FOR/NEXT loop is introduced in chapter 5. It is not emphasised that the FOR/NEXT structure is designed for looping situations in which the number of iterations is known in advance. In fact, it seems that the authors do not believe this is the case as they give three examples illustrating how to tamper with the structure, one of which is as follows:

```
10 FOR I = 1 TO 20
15 READ X
20 IF X = 0 THEN 50
25 IF X > 100 THEN 40
30 LET C = C + 1
40 NEXT I
50 (rest of program)
```

This is clearly not a FOR/NEXT situation at all as the loop is terminated either when I exceeds 20 or when X=0. The structure of this program fragment could be improved as in figure 5.7. This would result in the following code:

```
 10  LET I=1
 20  LET C=0
 30  READ X
 40  REM STARTLOOP
 50    IF (X=0) OR (I>20) THEN 100
 60    LET I=I+1
 70    IF X <= 100 THEN C=C+1
 80    READ X
 90    GOTO 40
100  REM ENDLOOP
```

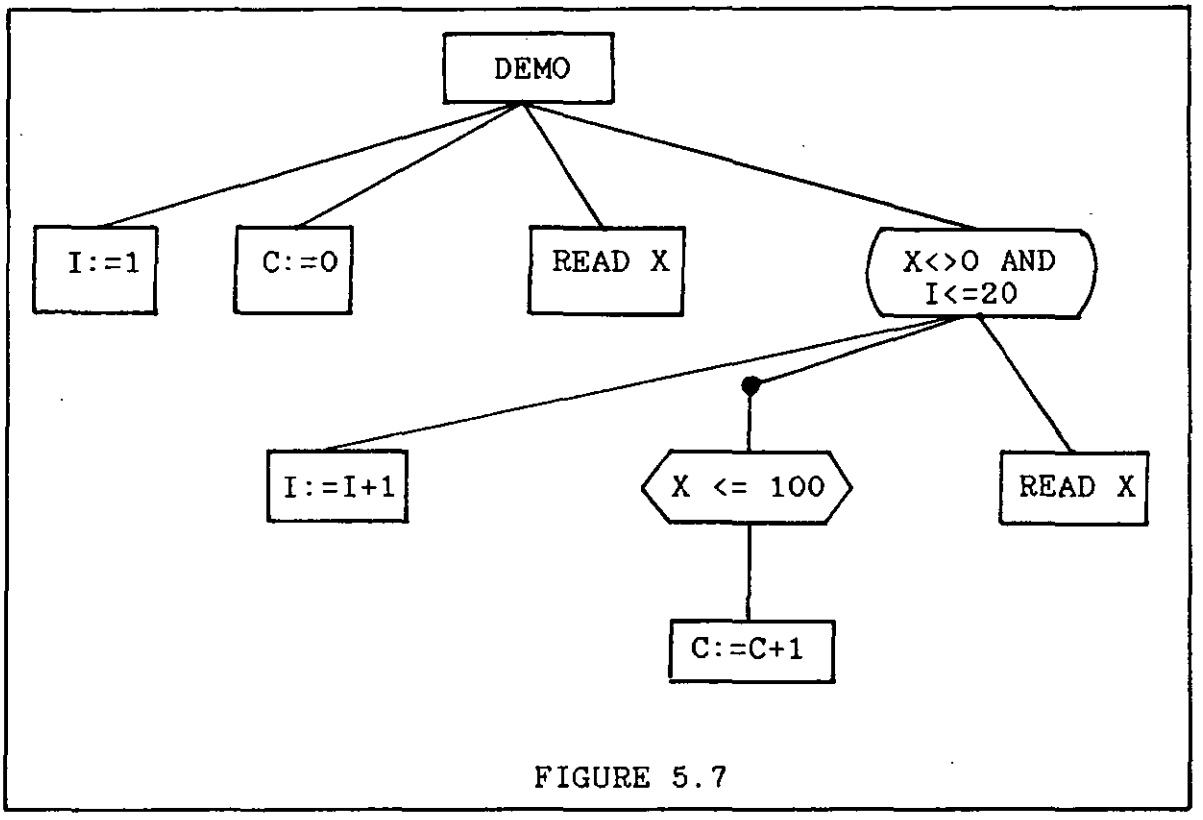


FIGURE 5.7

A further point to note about the listing from the book is that the authors fail to initialise the variable C. This practice is carried out throughout the book, the assumption being that uninitialised variables will have zero value. While this is true for most BASIC implementations, it is not good practice and would not work with most other languages.

No problem specification is given for this program fragment, or for the other ones which tamper with the FOR/NEXT structure, and no explanation is given for coding them in this way. The

motivation seems to be to demonstrate not only how the statements of BASIC work but also to show how they may be 'fooled'. The student, however, is left with the impression that programming consists of mysterious tricks.

Boolean operators (AND, OR, NOT) are introduced (p.58) as "additional facilities". A truth table is shown for each operator and some trivial examples are given. These examples are used merely to illustrate the truth tables and none are in solutions to stated problems. It seems that the authors do, in fact, regard boolean operators as "additional facilities" as they do not use them in their own programs. If, however, it is realised that selection and iteration are the fundamental structures in all programs, and if both of these structures are 'guarded' by boolean expressions, then these expressions and their construction with boolean operators are of vital concern.

5.2.3 'ADVANCED' BASIC.

Towards the end of the book there is a section (Chapter 15) on 'Advanced' BASIC. This is not, as one might expect, a chapter about solving very difficult problems with BASIC but one in which additional (and somewhat rare) BASIC statements are described. This emphasis on language 'facilities' at the expense of problem-solving is, however, consistent with the general approach of the book. These 'advanced' facilities are:

A. The IF/THEN/UNLESS STATEMENT (p. 264)

B. The FOR.....UNTIL STATEMENT (p. 266)

C. The FOR.....WHILE STATEMENT (p. 266)

These are merely looping and branching statements and can easily be constructed using simpler, more common statements. The

impression given by this chapter is that, to become an 'advanced' programmer, it is merely necessary to learn how more statements work. This is not so, as being a good programmer involves analysing problems in a disciplined, systematic way and detailed knowledge of numerous programming language statements is not much help in this regard.

This book was considered to be unsuitable for the purposes of the course, for the reasons outlined above.

5.3 FOUNDATIONS IN COMPUTER STUDIES WITH COMAL [49]

The preface of this book contains six stated aims, four of which concern 'general' computer appreciation. The other two are:

1. To develop an appreciation of structured problem-solving.
2. To develop skill in COMAL programming.

Most of the book is devoted to these two aims. The first two chapters fail to make a clear distinction between problem-solving, editing and disk management. These are all introduced together without any attempt to emphasise key concepts such as variables, assignment statements and input statements.

5.3.1 SELECTION

The first decision structure introduced is the IF/THEN/ELSE structure. This is represented in the structure diagrams by a decision box containing a boolean expression with two branches leading out of it. These branches are marked YES and NO respectively. The branch marked YES is executed if the boolean expression is TRUE and the branch marked NO is executed if it is

FALSE. This arrangement often leads to situations in which it is necessary to go the bottom of a diagram before accomplishing a task at the first level, as the author concedes in the example (on p.42) concerning conversion of singular nouns to plurals (see figure 5.8).

In the alternative diagram (figure 5.9) all the decisions, which are of equal importance, are represented at the same level of the diagram. This is a neater, easier to follow representation of the solution. This method is also more conducive to 'Top-Down' thinking as the decision is taken at a certain level of the diagram and does not intrude on lower levels if there are any.

This author's solution is then coded in COMAL (on p.43), using four nested IF/THEN/ELSE structures. On the following page it is remarked that the "need for multiple selection arises fairly often" and an IF/ELIF coding of the same problem is given. This is a simpler, clearer coding as there is really only one multi-way decision to be made, rather than four separate two-way decisions. This is reflected much more clearly by the alternative diagram presented here than by the diagram in the book.

All the remaining problems in the chapter involve multi-way decisions and are treated in either of two ways:

1. Solved using nested two-way decisions and coded using IF/ELIF.
2. Solved using one multi-way decision (like the one suggested here) and coded using the CASE structure.

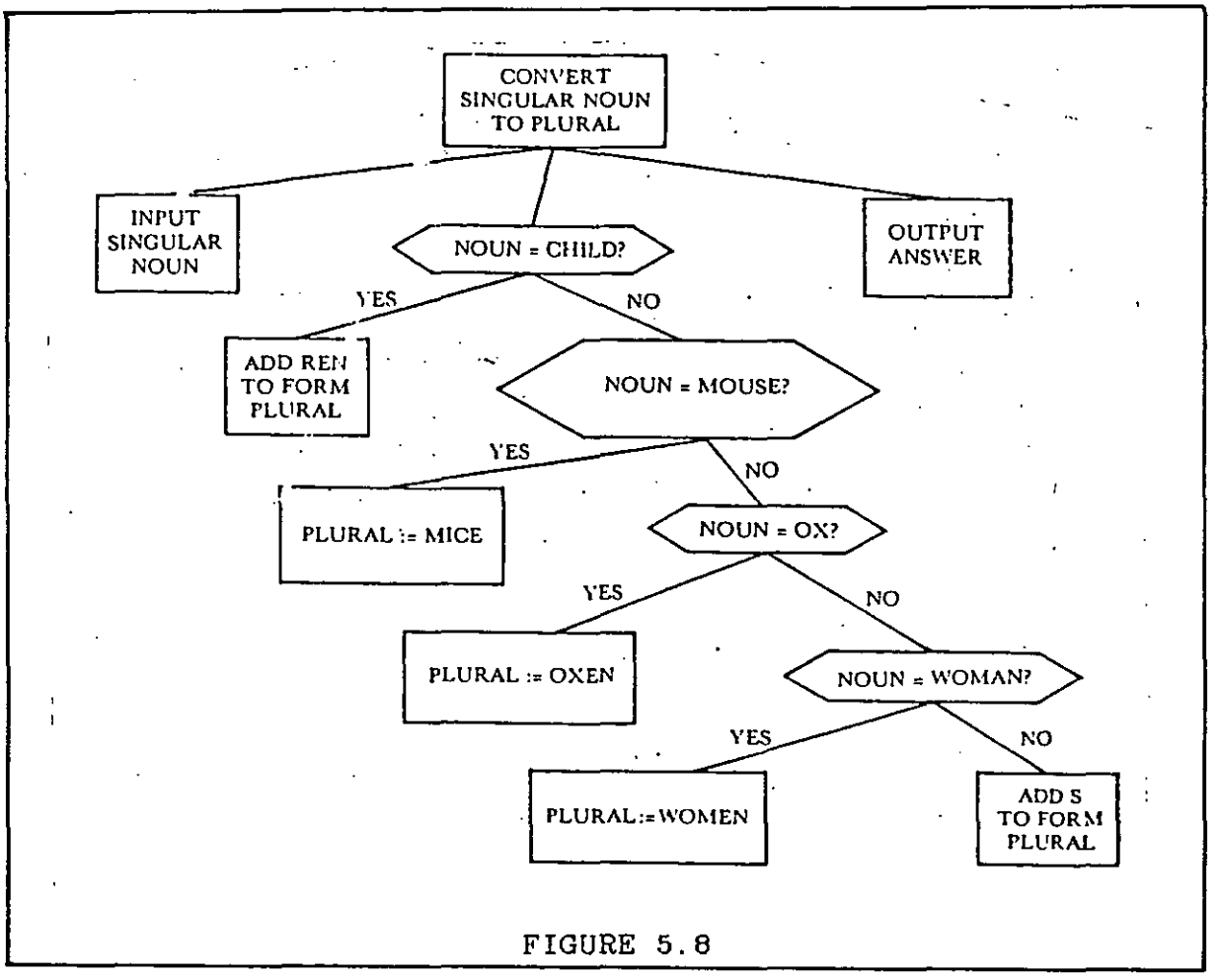


FIGURE 5.8

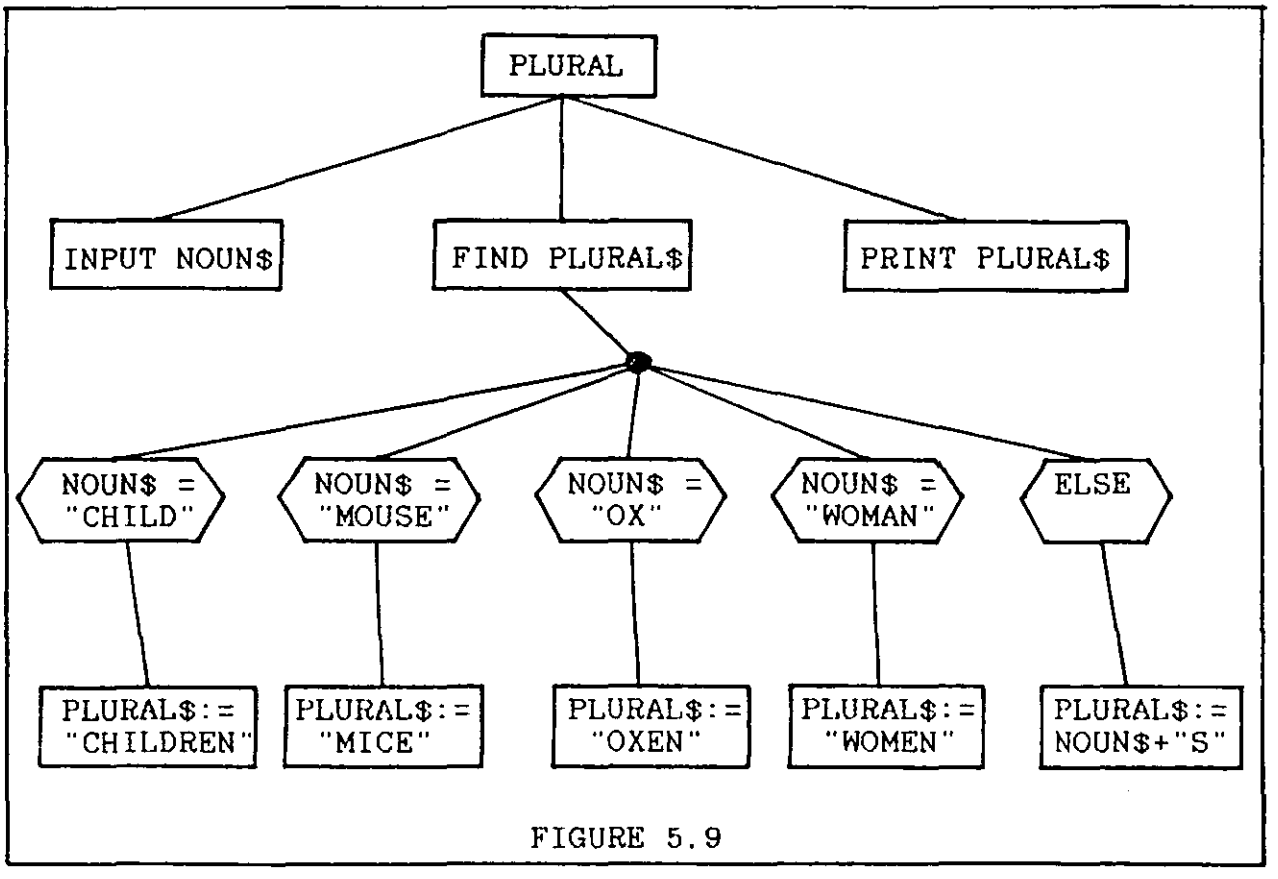


FIGURE 5.9

Towards the end of the chapter (p.55), a problem is shown to illustrate how the CASE statement can be very clumsy. The program is then coded using the IF/ELIF structure to show how it is superior to the CASE structure. In all, three different methods of making decisions are shown. This is unhelpful, especially when it is conceded that in certain circumstances two of these may be very unsuitable. This may leave students confused about which structure to use and will almost certainly lead them to employ mixtures of these structures when dealing with larger problems. It would seem that the author was more concerned with giving a full description of COMAL and less concerned with developing a structured approach to problem-solving.

5.3.2 ITERATION

Chapters 4, 5 and 6 are concerned with looping and introduce the REPEAT/UNTIL, FOR/NEXT and WHILE/ENDWHILE structures respectively. The first problem introduced to demonstrate the REPEAT/UNTIL loop is to find the mean of a list of positive numbers which are input. A negative value is used to indicate the end of the input. The diagram for the solution (figure 5.10) does not clearly suggest the structure of the solution and gives no indication of how the problem was analysed. It merely presents the statements in the same way as a program listing would. An alternative using the Top-Down method is suggested (figure 5.11).

The solution in the book (figure 5.10) is not strictly correct as it does not take account of the fact that there may be no positive input. This is a general feature of solutions that

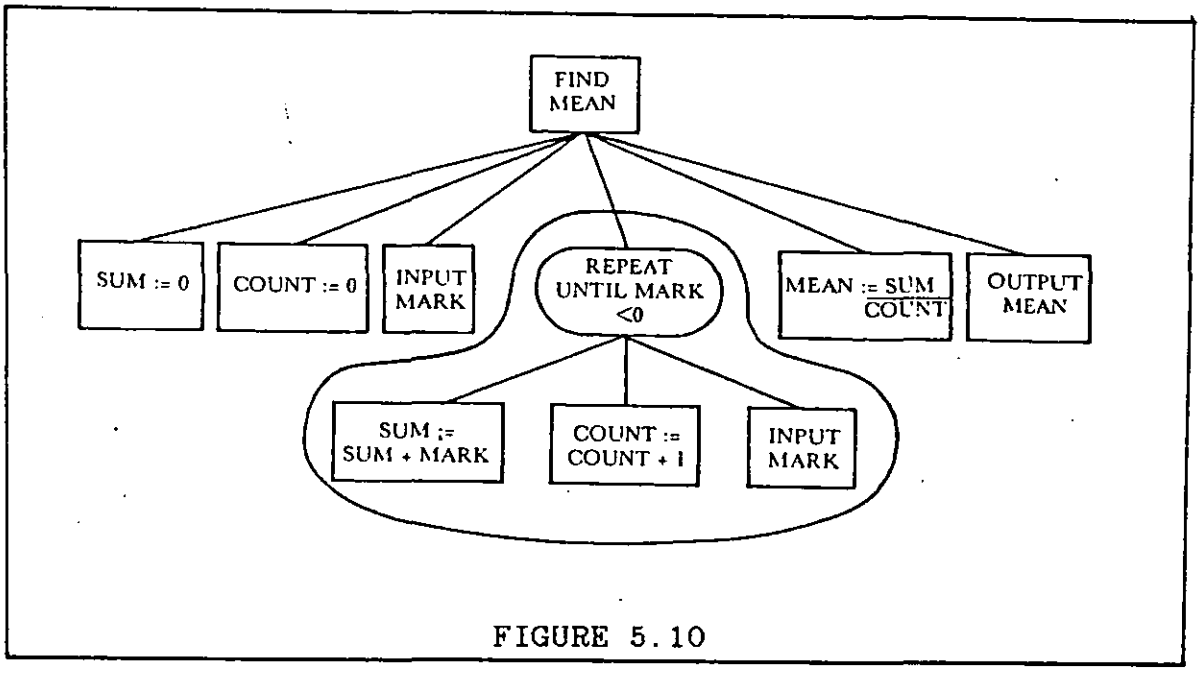


FIGURE 5.10

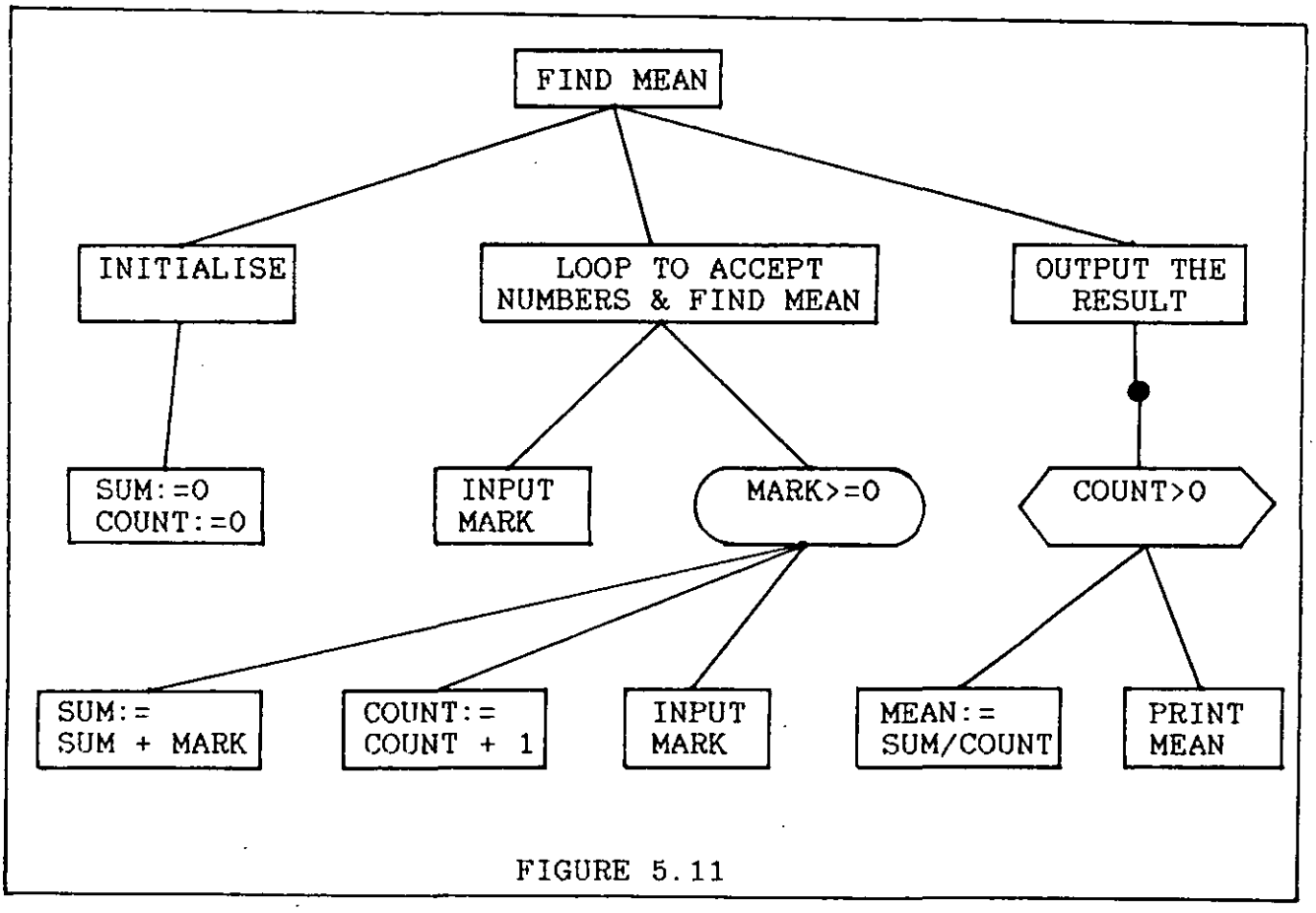


FIGURE 5.11

employ the REPEAT/UNTIL structure. In very many problems there are circumstances in which the actions inside the loop should not be executed at all. To guard against this, the boolean expression controlling the loop must be placed at the start of the loop. This means that the WHILE/ENDWHILE structure should be used.

Another problem dealt with in this chapter is to find the value of 7!. The program is as follows. (All REM statements have been removed).

```
 70  FACTORIAL:= 1
 80  N:= 7
 90  REPEAT
100    FACTORIAL:= FACTORIAL * N
110    N:=  N - 1
120  UNTIL N = 0
130  PRINT "7! = "; FACTORIAL
140  END
```

It may be noted that it is not necessary to write a program to calculate 7!, as this can be done with a simple print statement, i.e. PRINT 7*6*5*4*3*2*1. It must therefore be assumed that the author intended this program to be generalised to find N!. The given program is correct and does calculate 7! ; but, if it is generalised to calculate N! (where N is input), it will fail when N is given the value ZERO. The reason for this is again due to the nature of the REPEAT/UNTIL structure. The loop is not guarded properly and entry is allowed when N has ZERO value. This would lead to the loop being executed infinitely, because N is decreased inside the loop, and so "UNTIL N = 0" can never become TRUE.

It is just as simple to write a program for this problem which will work under all circumstances, assuming that the input is

valid (which is a separate issue). This of course involves a properly guarded loop and the WHILE/ENDWHILE structure.

```
10  INPUT N
20  FACTORIAL:= 1
30  WHILE N>1 DO
40    FACTORIAL:= FACTORIAL * N
50    N:= N-1
60  ENDWHILE
70  PRINT FACTORIAL
```

The loop is not executed at all if N=0 (or if N=1) and so FACTORIAL retains its original value of 1, which gives the correct answer for both 0! and 1!.

The FOR/NEXT loop is introduced in chapter 5. It is emphasised that this structure is only suitable where the loop is to be executed a fixed number of times. The 7! problem is recoded using a FOR/NEXT loop but still cannot be generalised to find N! as it contains the line:

```
90  FOR COUNT:= 7 DOWNTO 1 DO
```

If this were generalised it would be:

```
90  FOR COUNT:= N DOWNTO 1 DO
```

It is not clear what would happen here if N had the value 0 or 1 as both '0 DOWNTO 1' and '1 DOWNTO 1' are meaningless.

The WHILE/ENDWHILE loop is introduced in chapter 6. It is remarked (on p.88) that "the WHILE loop is more general than the REPEAT loop" and that "every REPEAT loop could be recast as a WHILE loop, but not the other way round". The author does not point out that the same may be said of the FOR/NEXT loop. (This is demonstrated in chapter 4 of this thesis). The use of the REPEAT loop is then justified because it is "often a more natural way to express iteration and may make programs easier to

read". It is not clear what "natural" means in this context but it is probably fair to assume that it is concerned with clarity of expression. No examples are given to illustrate this claim. In another example (on p.138) the author uses WHILE/ENDWHILE for a fixed iteration loop. While this is perfectly correct, it is likely to lead to confusion for students who have been told that the FOR/NEXT loop is the one to use under these circumstances.

The treatment of both selection and iteration involves too many programming constructs and so distracts from the most important issue, i.e. problem-solving. All the problems in these chapters can be solved using the WHILE/ENDWHILE loop and in many cases the solutions are more general and clearer. Even if all the other loops are to be introduced, the WHILE/ENDWHILE loop should be introduced first as it is the most general. It also encourages a Top-Down approach as the loop control condition is usually constructed before the body of the loop is written. This in turn encourages the student to construct the condition very carefully.

Boolean operators are almost totally ignored and are dealt with in less than half a page (p.141). This indicates that the author does not consider boolean operators to be important and, in fact, they are not used very much in subsequent programs. No attempt is made to show how these operators may be combined to build up powerful conditional expressions.

5.3.3 STRUCTURE DIAGRAMS

Throughout the book, structure diagrams are used to develop solutions before coding them in COMAL. A typical example,

involving the calculation of income tax, appears on page 24 (figure 5.12). This diagram is very wide and not very deep. This is an important characteristic of most of the diagrams in the book, indicating that a Top-Down method has not been used. The diagram in figure 5.12 is a direct restatement of the program listing and appears to have been derived from the listing rather than vice versa. The alternative solution (figure 5.13) gives a much clearer idea of how the problem was approached.

A later problem (figure 5.14) concerns finding the smallest positive power of 2 that exceeds a given (input) number. This solution contains no output statement and is obviously not a Top-Down solution. There is also an unnecessary variable (POWER) which makes the solution quite difficult to understand. The alternative (figure 5.15) is much simpler.

It would seem, from the examples presented here and the others in the book, that the diagrams are extremely 'close' to the coded COMAL programs. Because of this, they tell us little more than the program listing and so are of relatively little value.

While this book is closer to the type of course being developed than the previous one, it was considered unsuitable because of the lack of a proper Top-Down approach and the confusion caused by the presentation of all the looping and branching structures.

5.4 STRUCTURED PROGRAMMING WITH COMAL [50]

This book is intended to be a "suitable introduction to problem analysis and programming for a complete beginner or for someone who knows BASIC". It would seem, however, that the book is

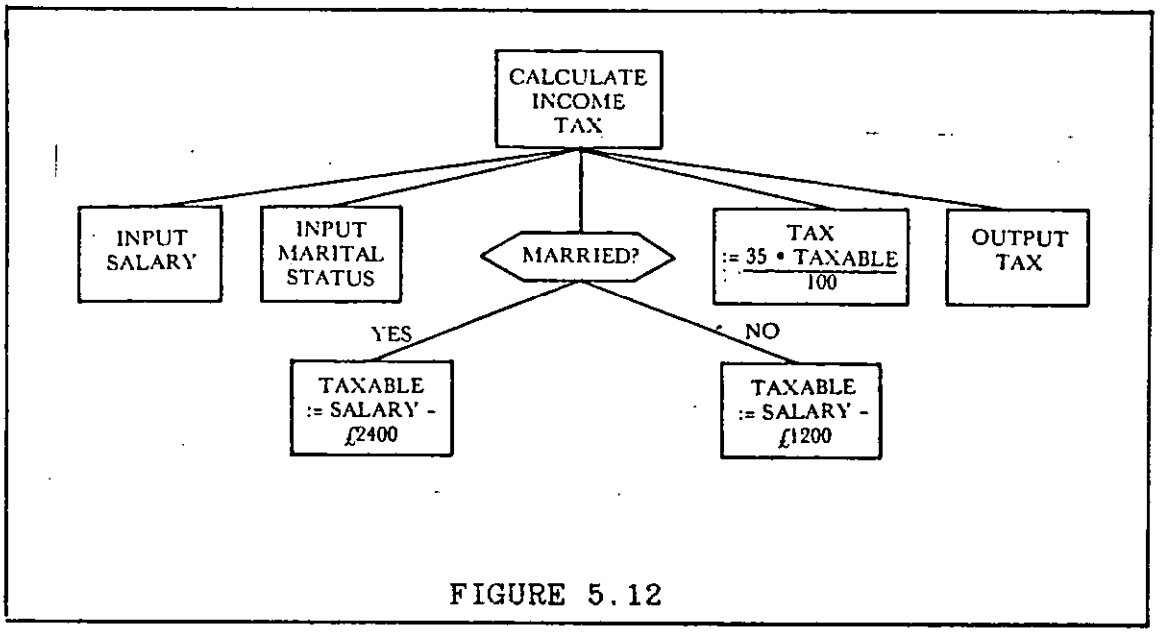


FIGURE 5.12

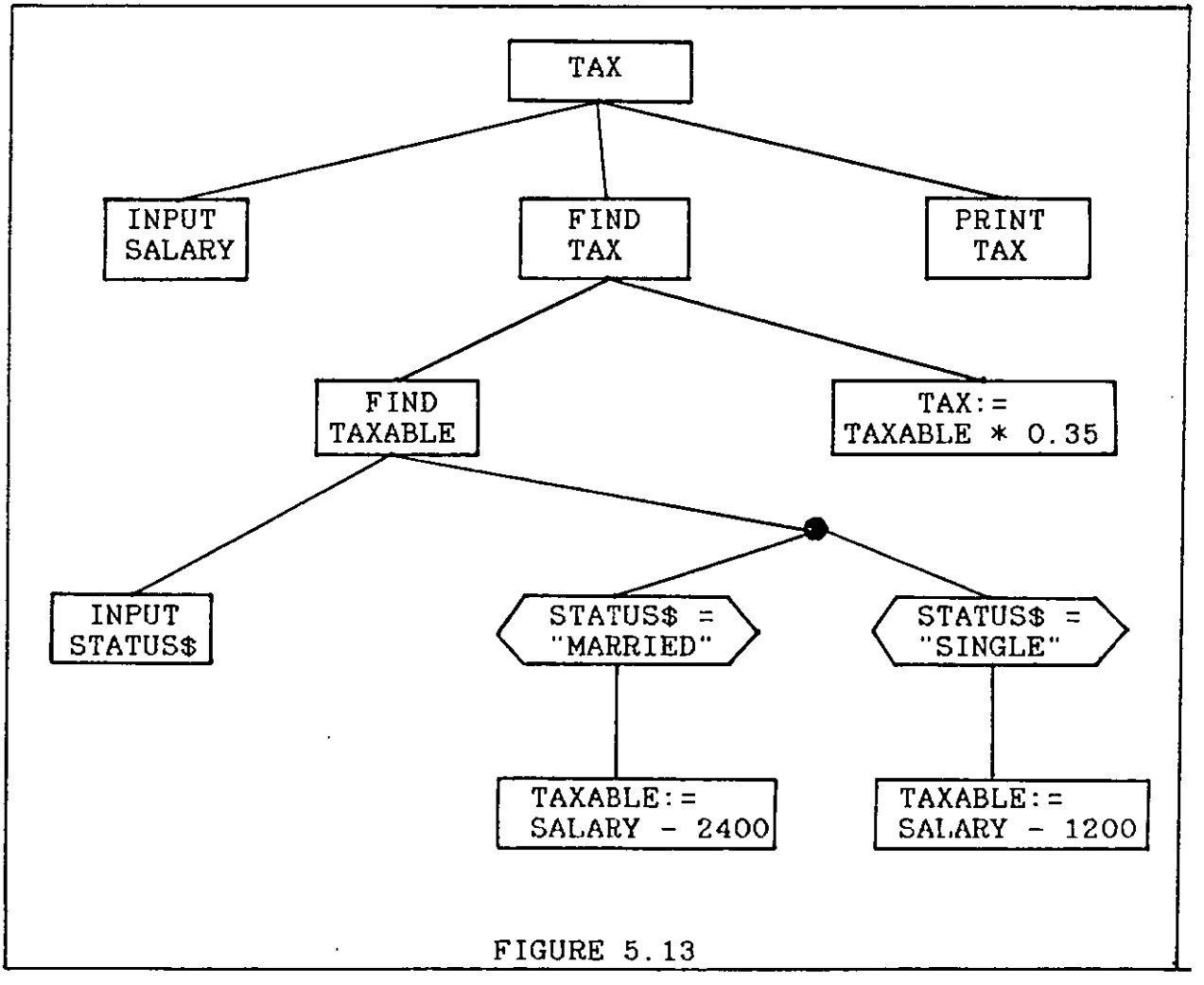


FIGURE 5.13

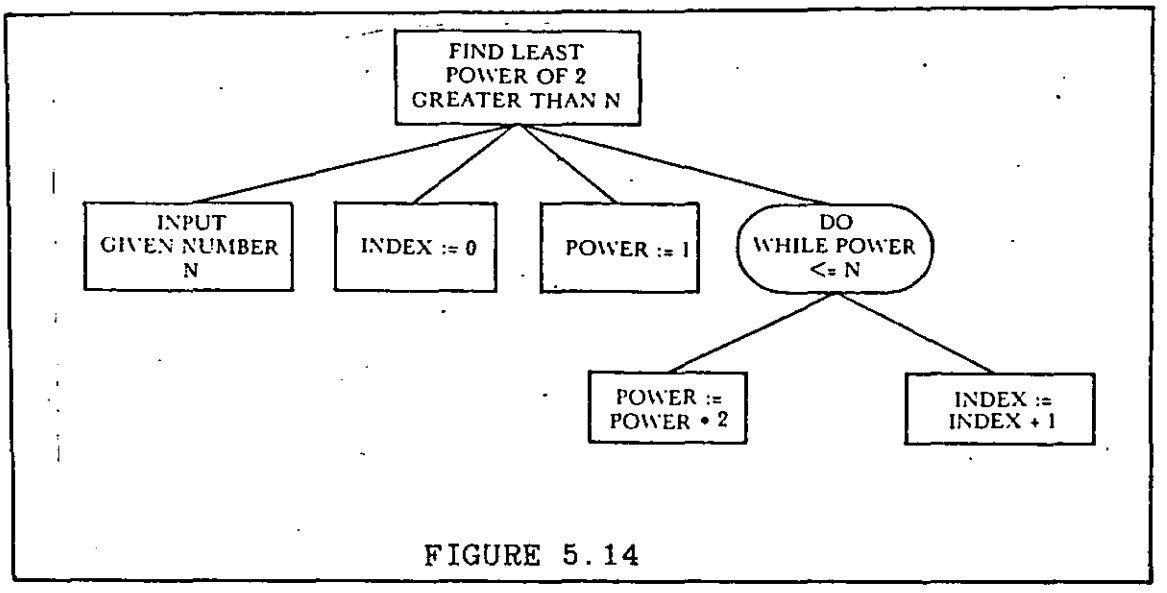


FIGURE 5.14

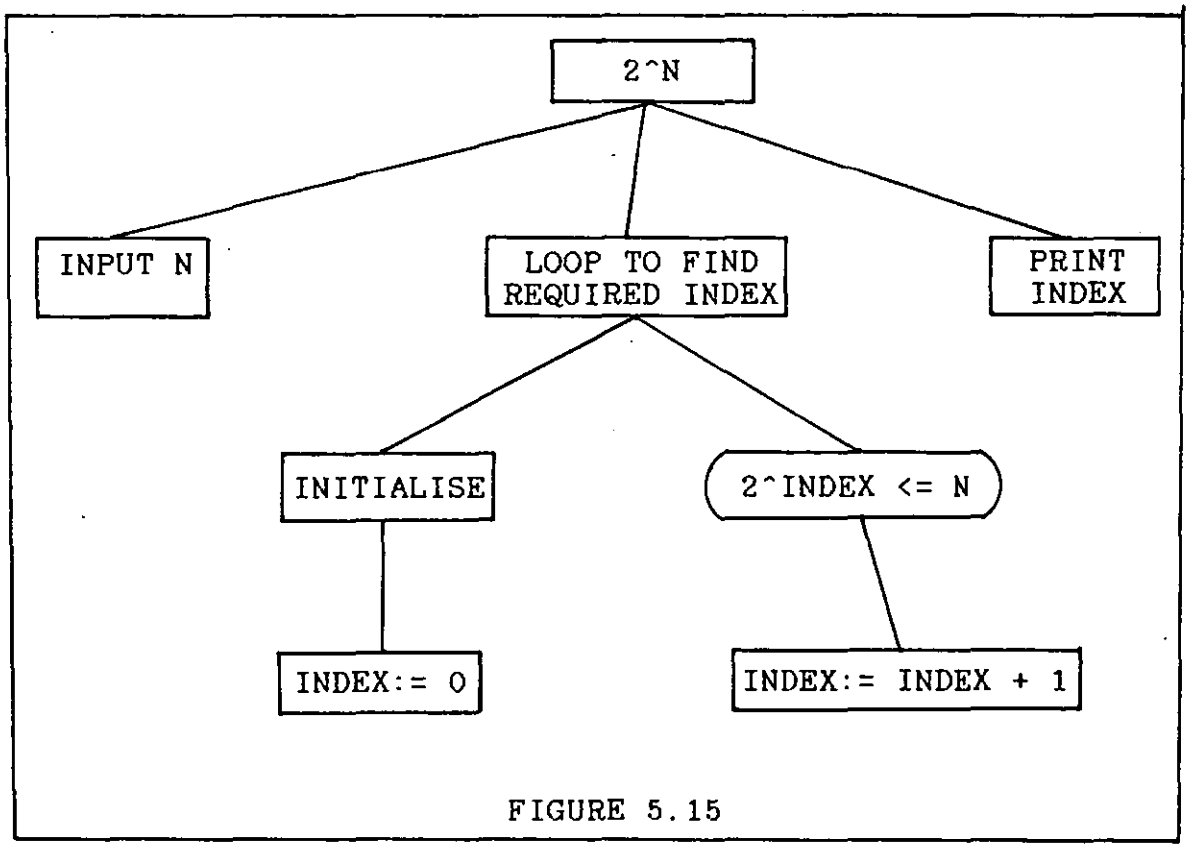


FIGURE 5.15

geared very much towards someone who knows BASIC, or some other programming language, reasonably well. The author assumes knowledge of many of the initial concepts and so there is a very skimpy treatment of variables and of the assignment statement. The examples given at this early stage concern both the use of string processing statements and the use of string variables to simulate records. This is not suitable for a complete beginner. Similarly, some of the sample programs given in the initial stages are extremely complex and often involve control structures and COMAL statements which have not yet been mentioned in the book (e.g. p.32). This tendency to use COMAL statements before they have been explained is a serious flaw and also occurs with conditional expressions (p.29), procedures (p.36) and boolean operators (p.61). In many cases new ideas are introduced with over-complex examples, e.g. conditional structure (p.58), CASE statement (p.72) and arrays (p.76).

5.4.1 ITERATION

The FOR/NEXT loop is introduced first, followed by the REPEAT/UNTIL loop. The author recognises that the REPEAT loop "has a defect" and warns of the danger of "an unexpected zero case" (p.122). Despite this, the REPEAT/UNTIL and FOR/NEXT constructs are used most of the time because "these structures often express the sense of what is being done in terms that people can more readily appreciate" (p.128). The WHILE/ENDWHILE construct, which is acknowledged as being "more general" and "safer", is reserved for special cases where no other construct is possible. It is strange that the FOR/NEXT and REPEAT/UNTIL constructs should be considered 'easier to understand' than the

WHILE/ENDWHILE. There is no evidence to support this and it is difficult to see how any one of the constructs should be more difficult to understand than the others.

5.4.2 SELECTION

The IF/THEN/ELSE construct is introduced first for two-way decisions and the CASE statement is used for multi-way decisions. The ELIF construct is introduced later for multi-way decisions "which do not convert naturally into discrete sets of values", i.e. for special cases. There is only one example (on p.130) to illustrate its use and it is only used twice in the remainder of the book.

Unlike the other textbooks reviewed, there is a reasonably detailed treatment of boolean operators, although de Morgan's Laws are not mentioned. However, the author also makes the distinction between inclusive and exclusive OR, which is, at best, irrelevant. Once the rules for constructing a truth table have been given, there is no need for this distinction, as it may lead to confusion. Indeed the author has confused himself on this very point and makes the misleading statement that "OR leads to a true compound statement if both simple statements are true" (p.63).

Throughout the book the author uses the 'post holes' problem to illustrate various points. It would have been preferable to have used problems that could actually be programmed on a computer. Most of the worked problems are analysed by listing a few points under the heading Problem Analysis/Program Design and this is usually followed directly by the coded program.

Structure diagrams are introduced but only used to solve three 'computable' problems in the whole book. The motivation for introducing the diagrams is not clear as the author does not use them himself.

This book is not suitable for the proposed course as it assumes a certain knowledge of BASIC, does not use structure diagrams in a desirable way and only uses the two most important and general control constructs (WHILE/ENDWHILE and IF/THEN/ELIF) for 'special cases'.

5.5 BEGINNING COMAL [52]

This book was written by one of the designers of COMAL and claims to be "not only about COMAL; it is also an introduction to structured programming in general". The book is accompanied by a disk containing numerous programs, listings of which are given in the book. The whole focus of the book is on showing how COMAL statements work, using the given programs. There are numerous exercises, most of which involve running and editing these programs. There are also numerous very trivial questions which merely ask the learner to supply the line-number of some particular statement or statements from the given listings.

Structure diagrams are used occasionally, but always to illustrate the structure of a program which has already been written. They are not used as tools for developing solutions but as devices for illustrating program listings by suppression of detail. The exercises which involve the diagrams all consist of 'filling in the blanks' in an incomplete diagram by checking the program listing.

The treatment of control structures is similar to that of the other COMAL books. For iteration, the FOR/NEXT loop is introduced first, followed by the REPEAT/UNTIL and finally the WHILE/ENDWHILE. In the supplied programs the FOR/NEXT and REPEAT/UNTIL structures predominate. For selection, the IF/THEN/ELSE construct is used for two-way selection and the CASE/ENDCASE construct is used for multi-way selection. Amazingly, the IF/THEN/ELIF structure is not mentioned anywhere in the book. As in the previous book, the treatment of boolean operators is very skimpy.

This book is not concerned with solving problems. All the material, exercises etc. are concerned with trying to understand, edit and modify given programs. All of these programs must be taken as 'given' as there is no problem specification for any of them. Because of this, the book has very little relevance to the course described here, although it may be of value to those seeking an understanding of the way in which COMAL statements work.

5.6 THE DESIGN AND USE OF STRUCTURED ALGORITHMS [7]

This book was issued to all second level schools in the country "as an aid to the teaching of computer studies". The aim was to "report on the design and use of structured algorithms". It is geared more towards teachers' needs than towards students' and assumes familiarity with the ideas of input, output and assignment. The first chapter describes the historical development of structured programming and this is followed by an excellent chapter outlining the general features of structured programming, including Top-Down design. The remainder of the

book consists of programming examples using structure diagrams. The appendix contains COMAL programs for each of the structure diagrams.

The authors state that they wish to "present structured programming as a method which is systematic and which produces easily understood and logical algorithmic solutions to problems". This is not achieved however, as the book contains major flaws which render it of little value:

1. Despite the excellent description of the Top-Down method, the problems are not solved in a Top-Down manner (apart from a few introductory 'non-computer' problems).
2. All of the COMAL looping and decision structures are used and there is no set of translation rules given to convert the diagrams into COMAL. The result of this is that many of the COMAL programs bear very little relationship to the diagrams from which they were derived.
3. In many cases diagrams and programs are given without any problem specification.

Two examples will illustrate these points:

1. The diagram in figure 5.16 is an algorithm for "searching a list of numbers for a target number". On page 34 it is stated that the first level of a solution should be "concerned with WHAT must be done rather than HOW it must be done". It is patently obvious that the first level of this solution does not describe what must be done and

is, in fact, totally meaningless by itself. Practically all the programming examples in the book are carried out in this style. This particular example also contains an unnecessary boolean variable. An alternative Top-Down solution using the proposed mini-language is given in figure 5.17.

2. A diagram is given (p.67) to solve the problem of converting a given exam percentage into a grade. The diagram and the associated program (p. 151) are in figure 5.18. While the supplied program does actually provide the same output as the diagram, it does so by a different process. There is nothing in the diagram which is equivalent to line 40 of the program. Neither do the values in the CASE structure bear any resemblance to those in the conditional statement in the diagram.

This practice occurs throughout the book and is a direct consequence of the lack of a set of translation rules for deriving the program from the diagram. There are no translation rules because it would be extremely difficult to devise a coherent set of rules to cover all the control structures used in the book.

The fact that the diagrams are badly organised, badly translated and not Top-Down means that this book, apart from the introductory chapters, adds nothing to our understanding of systematic problem-solving and structured programming.

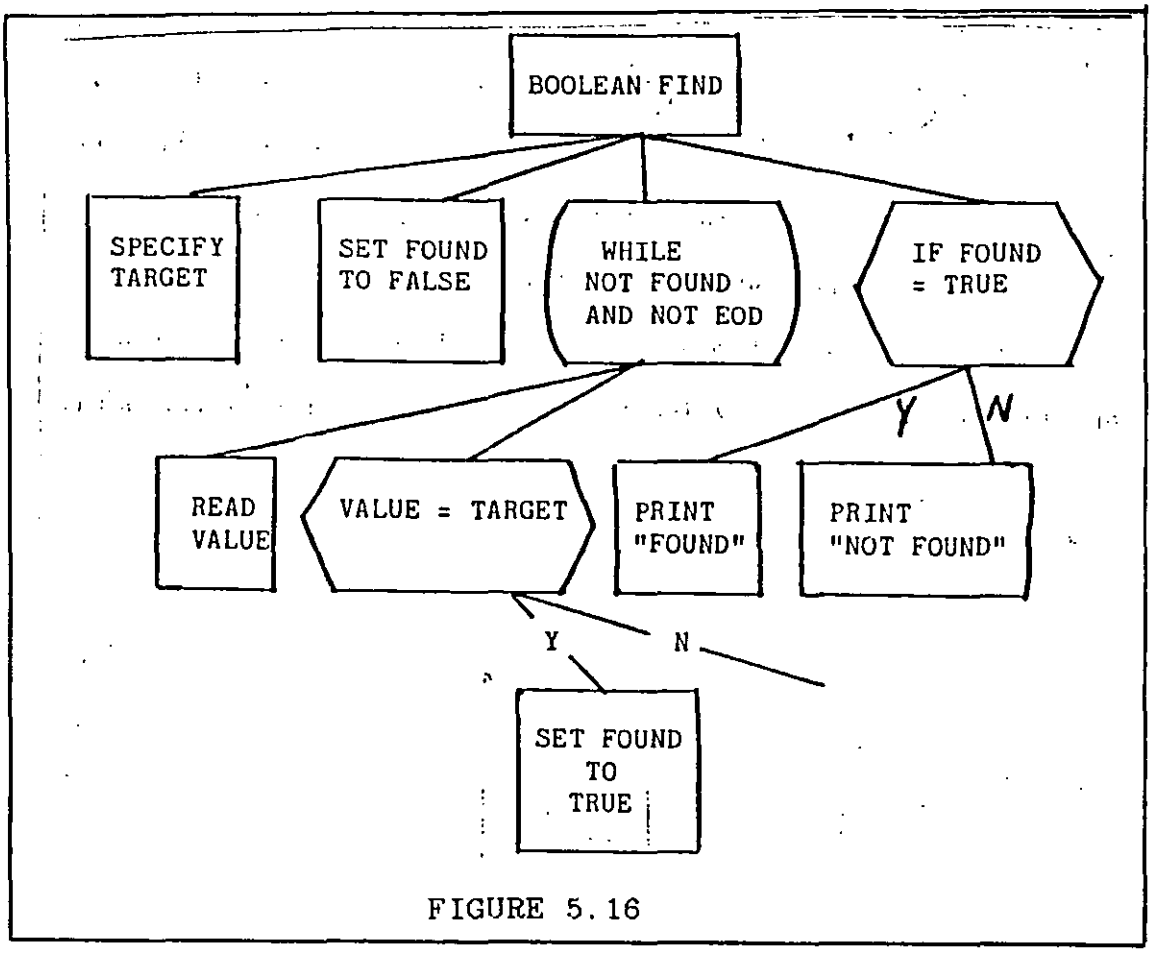


FIGURE 5.16

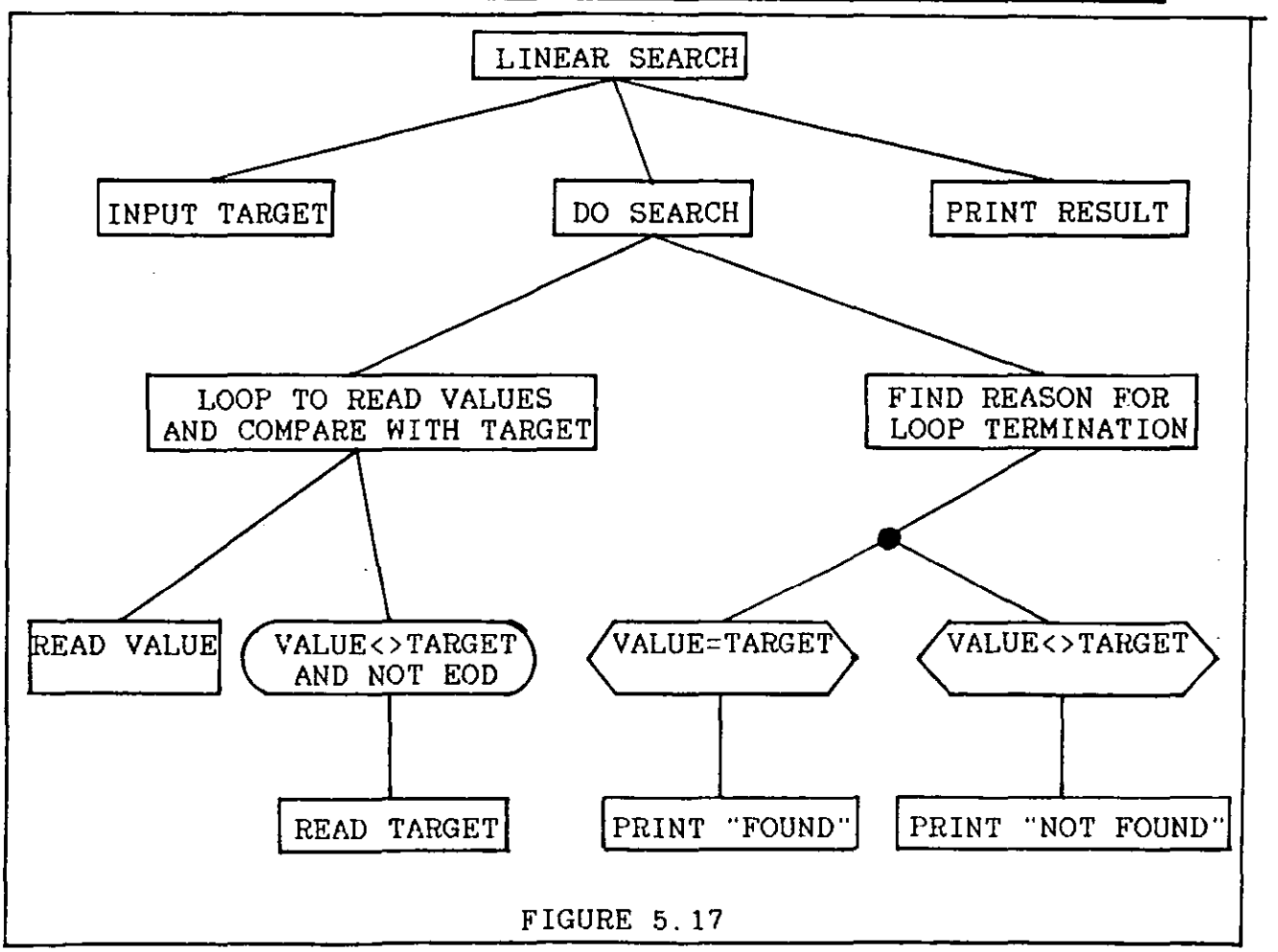
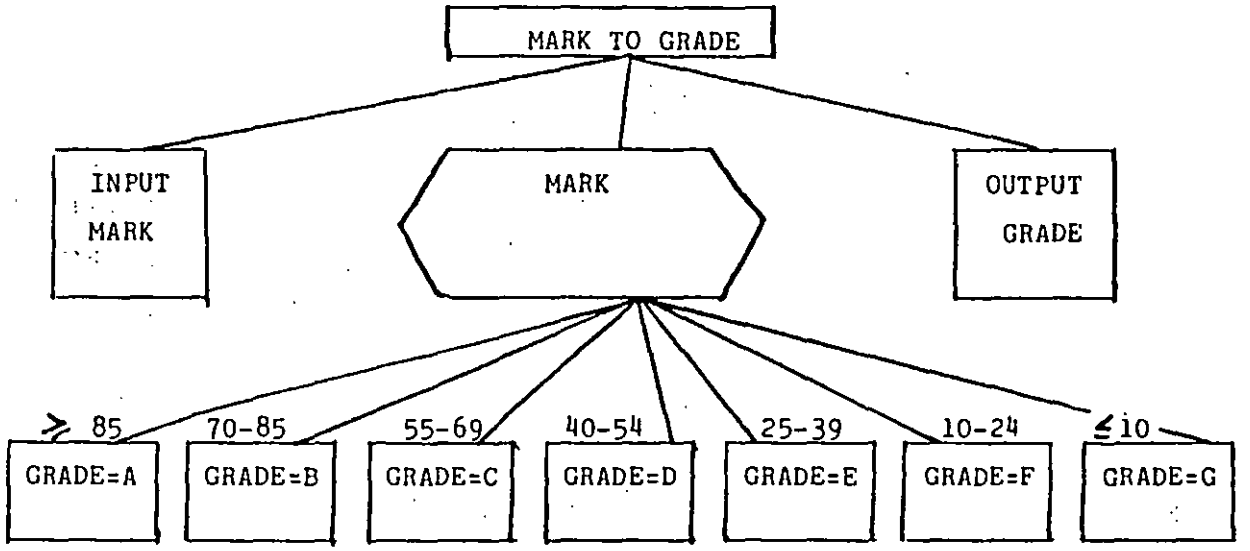


FIGURE 5.17




```
0010  //PROGRAM TO CONVERT PERCENTAGES TO GRADES
0015  //PROGRAM C 4.8
0017  CLEAR
0020  INPUT " ENTER MARK 0 TO 100 ": MARK
0030  //CONVERT MARK TO A VALUE IN RANGE 1 TO 8
0040  GRADE:=INT((MARK+5)/15)+1
0050  CASE GRADE OF
0060  WHEN 1
0070  PRINT "NG"
0080  WHEN 2
0090  PRINT "F"
0100  WHEN 3
0110  PRINT "E"
0120  WHEN 4
0130  PRINT "D"
0140  WHEN 5
0150  PRINT "C"
0160  WHEN 6
0170  PRINT "B"
0180  OTHERWISE
0190  PRINT "A"
0200  ENDCASE
0210  END
```

FIGURE 5.18

5.7 CONCLUSION

It would seem from the books reviewed here that the general practice in schools' programming courses is to emphasise the semantics of the chosen programming language, while paying relatively little attention to general problem-solving strategies. In all cases the authors used most of the available control structures, although there was a consensus that the WHILE/ENDWHILE and the IF/ELIF structures were the most general of those available. Despite this, these two structures were used much less often than the other looping and branching structures. Although structure diagrams were used in most of the books, they were not used in a proper Top-Down manner in any of them. It appears that structure diagrams are seen as devices for representing programs that have already been written, rather than as aids to problem-solving. It was possible to solve every problem in these five books using the mini-language and in the majority of cases the solutions were better, clearer and more general than those in the books. No book was found to be suitable for the proposed course but many interesting exercises were found and these were adapted for subsequent use in the course.

CHAPTER 6

PROJECT RATIONALE

6.1 SCHOOL AND STUDENT BACKGROUND

The course was taught over a two year period, '83/'84 and '84/'85. The initial course was implemented in each year with different groups of fifth year students (all aged 15 or 16 years at the start of the course) and the continuation course was implemented just once, with sixth year students. The courses will be referred to as the fifth year course and the sixth year course respectively.

The courses were implemented at the Holy Faith Secondary School, The Coombe, Dublin. This is a non fee-paying, girls' school run by the Holy Faith order of nuns. The school is situated in the Liberties of Dublin, an old and historic area close to the city centre, in which the school-going population is declining. The area is currently beset by many social problems such as unemployment, drug abuse and vandalism. Many of the children in the school have had close contact with such problems. The population of the school is decreasing rapidly, falling from 665 in '78/'79 to 400 in '84/'85. The pupils in the school are all from working class backgrounds. Many of them live in inner city apartment blocks and most of the rest commute from large council estates on the outskirts of the city. This latter group are mostly children of parents who originally lived in the vicinity of the school.

In general the girls in the school are not ambitious. Most are from large families (on average 4 or 5 children) in which there is no academic tradition. Neither is there any tradition of women working outside the home or of aspiring to a career. Very few of the girls go on to third level education. For a girls' school of its type there is a good range of subject options at senior level, although these are being curtailed as the school population drops. The number taking higher level papers at Leaving Certificate level in Mathematics and Science subjects is very small and falling:

Mathematics	approx.	2%
Physics	"	2%
Chemistry	"	10%
Biology	"	20%

The school acquired its three micro-computers, two Apples and one B.B.C., early in 1982. Dissatisfaction with a traditional, BASIC, programming course, taught in '82/'83, prompted the design of the present course. For this earlier course all students who applied were admitted. The majority of those who applied had no idea of what was involved and there was a very high dropout rate (60%). All of the students in the lower two streams dropped out so it was decided to seek applications for the present course from the two higher streams only. An informal selection procedure was used, based mainly on subjective evaluation by staff members but also taking note of performance in the Intermediate Certificate examination. The object was to choose the most able and hardest working students, as it was felt that the course, being a non-exam course, would require considerable commitment on the part of those

participating. Seventeen students were chosen in the first year and fifteen in the second. In each year two students dropped out, in all cases after just a few weeks. These students were among the weakest of each group, confirming that this type of course is best suited to more able students.

As very few of the staff have any interest in computing, the machines have not been used extensively apart from the programming courses. No 'computer culture' has developed as yet. In 1985 however, seven more computers were purchased and it is planned to use these extensively to teach programming, word processing, spreadsheets, databases etc. in the coming years.

In the first year of the course the machines were located in a large open room with virtually free access. The students developed a rota system for themselves. They divided into groups of two to five students and each group had priority access to the machines after school on one appointed day each week. They usually worked for about an hour, although there were cases of students being locked in because they became absorbed in their work . The machines were also available at lunch time (35 minutes) and morning break (10 minutes). They were also allowed access before school in the mornings and during any free classes that might arise due to teacher absence etc. These relatively free conditions meant that some students spent up to eight hours a week working at the machines, while the average was about one or two hours.

Unfortunately, due to security problems, access had to be restricted during the second year. The computers were relocated

in a small room beside the staff room and quite a distance from the students' own classroom. This room was kept locked at all times and even though the students were allowed access as in the previous year, they were now required to get a key from a teacher or from the office. This resulted in a dramatic decline in machine use, the only regular use being after school in the evenings. These after-school sessions were further restricted to forty five minutes duration.

Two forty-minute periods per week were allotted to each course. This gave a maximum of about forty hours instruction per year but this was never achieved due to half-days, teacher absence, house examinations etc. The actual time spent on each course was approximately thirty five hours. Attendance by the students averaged about 80%. For the first few weeks of the fifth year courses a good deal of class time was spent at the machines. As the students became more confident, less time was spent on practical demonstrations until, eventually, practically all instruction took place away from the machines. The main tools used to deliver the material were overhead projector slides and student handouts. Examples of these handouts are shown in appendix A.

In an earlier survey, carried out by their career guidance teacher, many of the students who undertook the course had expressed an interest in computer-related jobs. None of them, however, had any idea of what computing involved and had only a very vague grasp of the difference between, for example, the work of a programmer and the work of a computer operator. Many had acquired the idea that a computer was all-powerful and knew

everything. For these reasons, it was felt that learning programming would give them valuable insight into the way in which computers work and would also help them to develop self-confidence in their use.

It was also noticed from teaching mathematics in the school over a period of years that students were very much inclined to give up when faced with even moderately difficult problems. If the solution was not immediately apparent, the normal reponse, even from the more able students, was to accuse the teacher of not having taught them how to solve that particular type of problem. It was felt that this response was at least partly due to low self-esteem and low expectations. By using a very systematic Top-Down method that could be applied to numerous different problems, it was hoped that this problem could be overcome.

6.2 AIMS OF THE COURSE

1. To teach important concepts of modern algorithmic development including Top-Down design.
2. To introduce students to systematic problem-solving.
3. To develop an appreciation of the need for a systematic, structured approach to solving problems.
4. To give students practice in developing and validating their own ideas.
5. To develop the ability to make logical decisions and to discuss ideas intelligently.
6. To develop a positive attitude towards work and problem-solving.
7. To develop initiative, creativity and perseverance.
8. To develop the capacity to apply existing knowledge.

9. To develop consciousness of the learning process.
10. To develop an appreciation of the need to think and write clearly.
11. To build confidence in the students' own abilities.
12. To demystify the computing process.
13. To develop confidence in the students' ability to use and control a microcomputer.
14. To develop an appreciation of the precision of communication afforded by a computer language.
15. To develop an awareness of the capabilities and limitations of microcomputers.

6.3 OUTLINE OF THE SYLLABUS

Problem-Solving Concepts.

1. Top-Down design and stepwise refinement.
2. Procedural abstraction.
3. Algorithm representation in graphical form.
4. Methodology for approaching large problems.

Programming statements and structures.

1. The idea of a variable (both Real and String).
2. Input, Output and Assignment statements.
3. Read/Data statements.
4. Boolean expressions and operators. de Morgan's Law.
5. The conditional statement.
6. The iterative statement.
7. Procedures.
8. Some simple system functions (i.e. INT, RND etc.).
9. Arrays.

Coding Style.

1. Use of suitable variable names.
2. Use of suitable procedure names.
3. Use of indentation to emphasise structure.
4. Documentation techniques. Internal program comments and suitable screen prompts for the user.
5. The need for easily readable screen output.

Basic Computer Concepts.

1. Description of a computer system in terms of processor, memory and input/output devices.
2. Evaluation of programming languages.
3. Translation of programming languages into machine-usable form by assemblers, compilers and interpreters.

In developing a syllabus the most important elements were problem-solving concepts and programming structures. Coding style was also considered to be important because of its relevance to communicating clearly. It is impossible to work with a computer without some concept of how it functions and so a basic mental model of the computer system had to be established in the initial stages. The topics were not taught in the order listed above. The order is described in section 6.4. This syllabus is in accord with the report of the ACM Curriculum Committee Task Force, and is actually a subset of the syllabus recommended as a first course for Computer Science majors [27].

6.4 THE FIFTH YEAR COURSE

This section sets out the order in which the principal topics were taught and the objectives to be achieved through each section.

6.4.1 DEVELOPING THE CONCEPT OF A COMPUTER SYSTEM

The purpose of this section was to help the students to build a simple mental model of a computer system to enable them to carry out disk management and program editing tasks.

Objectives.

Students should be able to:

1. Start up COMAL when:
    a) The machine is OFF.
    b) The machine is running some other software.
2. Catalog a disk in either drive.
3. Load programs from disks in either drive.
4. List and run programs.
5. Use the system editing features (AUTO, RENUMBER etc.) as required.
6. Edit programs:
    a) Insert and delete lines.
    b) Correct syntax errors.
    c) Add and delete characters on any program line.
    d) Use the 'ESC' key as required.

6.4.2 VARIABLES, INPUT, ASSIGNMENT AND OUTPUT.

The overall purpose of this section was to develop the concept of a variable and to make the students aware of how variables

can be manipulated to solve simple problems involving input, assignment and print statements only.

Objectives.

Students should be able to:

1. Distinguish between valid and invalid COMAL variable names.
2. Distinguish between string and numeric variable names.
3. Write syntactically correct input statements, incorporating appropriate user prompts.
4. Write syntactically correct assignment statements for both string and numeric variables.
5. Write syntactically correct print statements in the following forms:
   a) Print a string constant.
   b) Print the value of a variable.
   c) Print the value of a variable with an explanatory prompt.
6. Find and correct syntax errors in short programs involving just these three types of statement.
7. Find and correct logical errors in such programs.
8. Trace the values of variables in such programs.
9. Predict the output from such programs for given input values.
10. Design algorithms using these statements to solve simple problems on topics with which they are already familiar (i.e. area, volume, profit & loss etc.).

11. Implement these algorithms on a machine.
12. Design clear screen displays and helpful input and output prompts for such programs.

6.4.3 STRUCTURE DIAGRAMS

The diagrams are essentially a tool for assisting students to analyse problems in a Top-Down manner. Therefore it was essential that they were able to construct and interpret the diagrams before any difficult problems were encountered.

Objectives.

Students should be able to:

1. Place a given set of statements into their correct positions in a diagram to solve a stated problem.
2. Write a list of statements in the correct order from a given diagram.
3. Fill in the blanks in a diagram describing a familiar process.
4. Draw diagrams to represent solutions to simple input/assignment/output type problems.
5. Find diagrammatic solutions for slightly more difficult problems involving three or four levels but requiring just these three types of statement.
6. Translate these solutions into COMAL.
7. Write COMAL programs from supplied diagrams.
8. Explain the overall meaning of more complex, teacher-supplied diagrams.

6.4.4 THE CONDITIONAL CONTROL STRUCTURE

This, the first control structure, is of extreme importance and should not be introduced until the students are very familiar with all the previous material. In both years this was towards the end of the first term.

Objectives.

Students should be able to:

1. Recognise the symbol used in structure diagrams to identify the start/end of conditional statements.
2. Recognise the boxes used in structure diagrams to contain the boolean expressions of IF structures.
3. Translate structure diagrams containing conditional expressions into COMAL.
4. Predict the output from such diagrams for given input values.
5. Place the correct statements from a supplied list into the appropriate boxes in a partially completed diagram for a specified problem.
6. Recognise situations in problem specifications that necessitate the use of a conditional statement.
7. Find and correct syntax errors in programs involving conditional statements.
8. Find and correct logical errors in such programs.
9. Use structure diagrams to solve simple problems requiring conditional statements, and translate these solutions into COMAL.
10. Evaluate simple boolean expressions for given values of the variables involved.

11. Evaluate complex boolean expressions containing boolean operators for given values of the variables involved.
12. Find the converse of any boolean expression.
13. Construct boolean expressions for conditions specified in words.

6.4.5 THE ITERATIVE CONTROL STRUCTURE

This, the second control structure, was introduced shortly after the start of the second term in each year. The idea of a loop is absolutely central to programming and so, as in the case of the other important concepts, considerable effort was devoted to establishing the important principles before considering more challenging problems.

Objectives

The students should be able to:

1. Recognise the symbol used in structure diagrams to indentify the start/end of a loop.
2. Translate structure diagrams containing loops into COMAL.
3. Trace the value of a variable through numerous iterations of a loop.
4. Trace the value of the boolean expression controlling a loop through numerous iterations of the loop.
5. Predict the output, for given input values, from a program containing a loop.

6. Place the correct statements, from a supplied list, into the appropriate boxes in a partially completed diagram for a specified problem involving a loop.
7. Recognise situations in problem specifications that necessitate the use of a loop.
8. Distinguish between situations which require 'fixed' iteration loops and those which require 'indefinite' iteration loops.
9. Find and correct syntax errors in programs involving loops.
10. Find and correct logical errors in such programs.
11. Write the general outline for any program involving N iterations of a loop, i.e. 'fixed' iteration.
12. Write the general outline for any program which uses a loop to add (or multiply) a list of numbers.
13. Solve simple problems involving loops and translate these solutions into COMAL.

6.4.6 PROCEDURES

The purpose of this section was to teach the students how to approach larger problems than had been considered so far and to impress on them the importance of the internal organisation of solutions to such problems.

Objectives

The students should be able to:

1. Recognise the symbol used in structure diagrams to represent a procedure.
2. Translate structure diagrams containing procedures into COMAL.

3. Predict the flow of control in COMAL programs containing procedures.
4. Recognise situations in problem specifications in which the use of procedures would be advantageous.
5. Find and correct syntax errors in COMAL programs containing procedures.
6. Find and correct logical errors in such programs.
7. Use the ENTER command to utilise library procedures supplied on disc.
8. Use the LIST command to store their own library procedures on disc.
9. Write the general outline for any menu-driven program.
10. Solve problems using procedures.

6.5 THE SIXTH YEAR COURSE

A group of six girls who had completed the fifth year course, and who had expressed an interest in pursuing the subject further, took part in the sixth year course. As the Apple machines were being used by the second group of fifth years, a COMAL ROM chip was acquired for the B.B.C. machine which was then exclusively available for the sixth years. This arrangement was not without its drawbacks as the COMAL ROM was a pre-production model with no documentation whatsoever. This was not a very serious problem as only a subset of COMAL was used and this subset did not differ very much from the Apple version. The lack of a disk drive was a much more serious problem and even though one was ordered at the beginning of the year it was not delivered until near the end of the first term. As all the

students were preparing for the Leaving Certificate examination, they had very little time, apart from the two scheduled classes per week, to work at this course. For this reason, it was decided that most of the classes would be of a workshop nature with the minimum of formal instruction. The students generally worked on problems in small groups and the teacher was available to advise at any time.

The overall aim of this course was to allow the students to further develop the skills that had been acquired in the first year and to give them confidence in their own problem-solving abilities. It was originally planned to run the course for the complete year and to cover the topics of arrays and files, as well as giving them some experience in the use of spreadsheets, databases, wordprocessors and other packages. However, the pressure of their examination work in other subjects forced the termination of the course midway through the year, and ruled out most of these topics. Eventually, the only topic covered was arrays.

6.5.1 ARRAYS

Arrays were the central focus of the sixth year course. The concept of an array is vital but was considered to be too difficult to be dealt with in the limited time available for the fifth year course.

Objectives.

The students should be able to:

1. Recognise situations in which the use of an array is appropriate.

2. Dimension both numeric arrays and string arrays correctly.
3. Write simple programs to READ values into an array and to PRINT them out.
4. Find and correct syntax errors in programs involving arrays.
5. Solve simple problems involving the use of arrays, including the linear search of an array.
6. Describe the bubblesort algorithm for an array.

6.6 CONCLUSION

The principal difficulties of the target group of students were identified as lack of confidence and motivation in relation to mathematical and scientific subjects, in addition to an almost total lack of appreciation of the whole computing process. Despite this, a number of students had expressed interest in computers and related careers. Therefore, topics were chosen, and ordered, to meet both the perceived needs of the group of students and to comply with the most recently recommended norms of introductory College Computer Science courses.

CHAPTER 7

IMPLEMENTATION OF THE COURSE

7.1 DEVELOPING THE CONCEPT OF A COMPUTER SYSTEM

Each student was given a sheet (appendix A, p.206) containing instructions for starting up COMAL. This was demonstrated at the keyboard and the students were then encouraged to carry out the same procedure in small groups. After describing the Apple keyboard an attempt was made to establish the mental model of the system shown in figure 7.1. This model was discussed using an overhead projector and then reinforced by requiring the students to load and run a number of short, teacher-written programs.

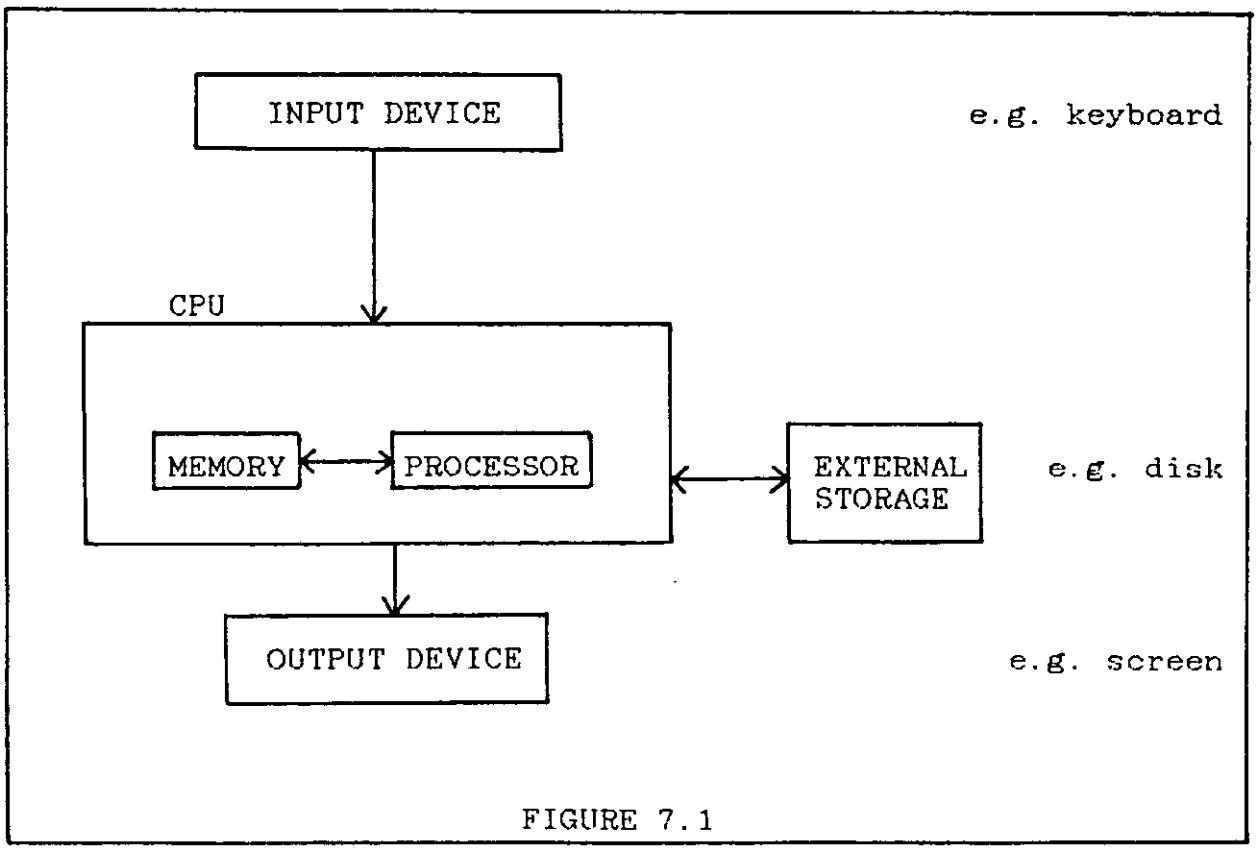


FIGURE 7.1

A distinction was made between immediate execution mode and deferred execution mode. Immediate commands are executed directly by the processor, while deferred commands are stored in memory and are only executed when required. Some of the simple COMAL direct commands were demonstrated (PRINT, CLEAR) and then a short sample program which merely printed a name and address on the screen was keyed into the computer. This was then used to illustrate the commands RUN, LIST and NEW. It was also used to demonstrate how extra lines could be added to the 'middle' of a program and how a line of a program could be changed by simply retyping it. Instructions for carrying out some other simple editing tasks were also supplied on the students' sheets and these methods were all demonstrated in class.

After the first few sessions the students were asked to write ten very simple programs, (appendix A, p.207) all of which only involved the CLEAR and PRINT statements, and to implement these on the machines. These were done over a period of two weeks and were completed very satisfactorily by all the students.

The limited editing skills that had been acquired by now were sufficient to allow the students to work at the machines unsupervised. The section on variables (see 7.2) was completed before dealing with disk management and more sophisticated editing skills. Lists of these more advanced disk and editor commands were drawn up on two reference sheets which were given to each student. For the sake of clarity, ALL the formats for each editor command were listed alongside an explanatory note,

rather than the more terse but common format adopted by the user manual:

i.e. LIST 20 .....line 20 is listed

LIST 20,50.....lines 20 to 50 are listed etc.,

rather than,

LIST [<start>] [,<end>]

This was done because it was felt that the majority of students would have difficulty with such an unfamiliar format, that it added to the air of mystery that surrounds computers in the first place and that each variation would have to be explained regardless. All of the editor commands were demonstrated on a machine, as were all of the disk commands with the exception of LIST/ENTER. (This is an alternative to SAVE/LOAD by which files are stored as a string of ASCII characters and can be recalled from disk without disturbing the contents of memory. The principal purpose of this feature is to allow for the retrieval of library procedures which may be used in numerous programs. This is not relevant to the early part of the course).

No specific exercises were given on the use of these commands but evaluation was carried out by informal observation of the students working at the computers. Very few problems arose in this area, indicating that all students were able to save, load and edit programs without difficulty. Whether they were using the editor in an efficient manner and using all of the editing facilities is another matter and may require some further investigation.

7.2 THE CONCEPT OF A VARIABLE

The idea of a variable is the first crucial programming concept and must be well understood before progressing to conditional and iterative statements. Variables were described as boxes in the computer's memory which have two characteristics, a name and a value (figure 7.2).

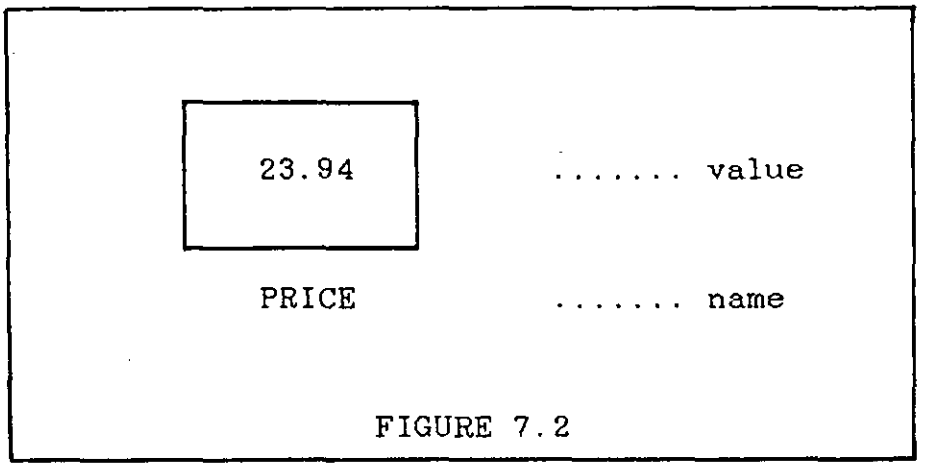


FIGURE 7.2

The rules for valid variable names were given to each student (appendix A, p.210). After some discussion of these rules, many examples of valid and invalid variable names were displayed on an overhead projector and the students were required to state whether each one was valid or not.

7.3 THE ASSIGNMENT STATEMENT

Assignment was described as 'putting values into boxes' and 'giving values to variables'. A detailed description of the syntax and semantics of the assignment statement was given to each pupil (appendix A, p.210). It was emphasised that the left hand side must contain only a variable name and that the right hand side must contain an expression whose value is then given to the variable on the left. It was pointed out that all variables in the right hand expression must have some initial value or else the whole statement would be meaningless. After

some examples which illustrated these points, numerous programs containing only assignment statements were displayed on an overhead projector. The students were required to calculate the value of each variable at the end of each program. They were instructed to set up a named box for each variable and to write its value inside the box (figure 7.3).

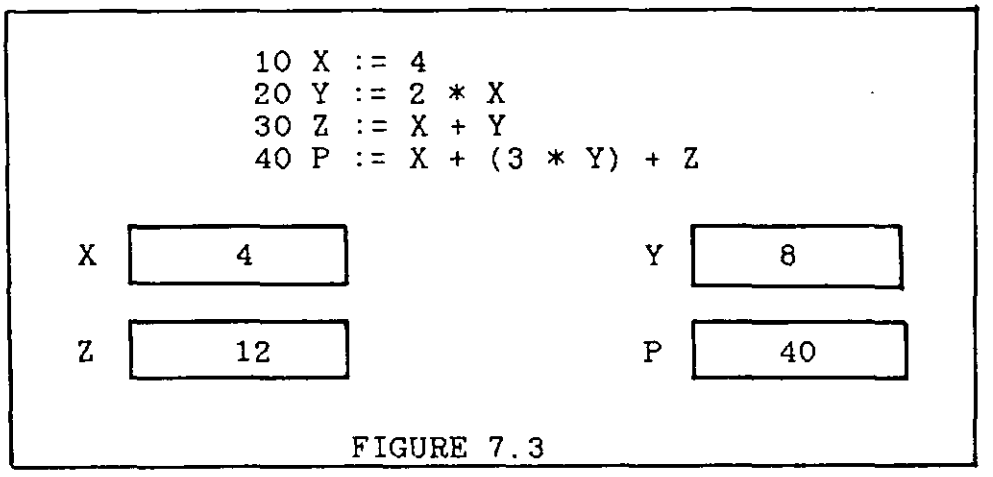


FIGURE 7.3

Slightly more difficult examples were then used in which the values of the variable were changed by the program (figure 7.4). This was to reinforce the idea that a variable can only have one value at a time. In these cases the old value was crossed out and the new value was written beside it in the box.

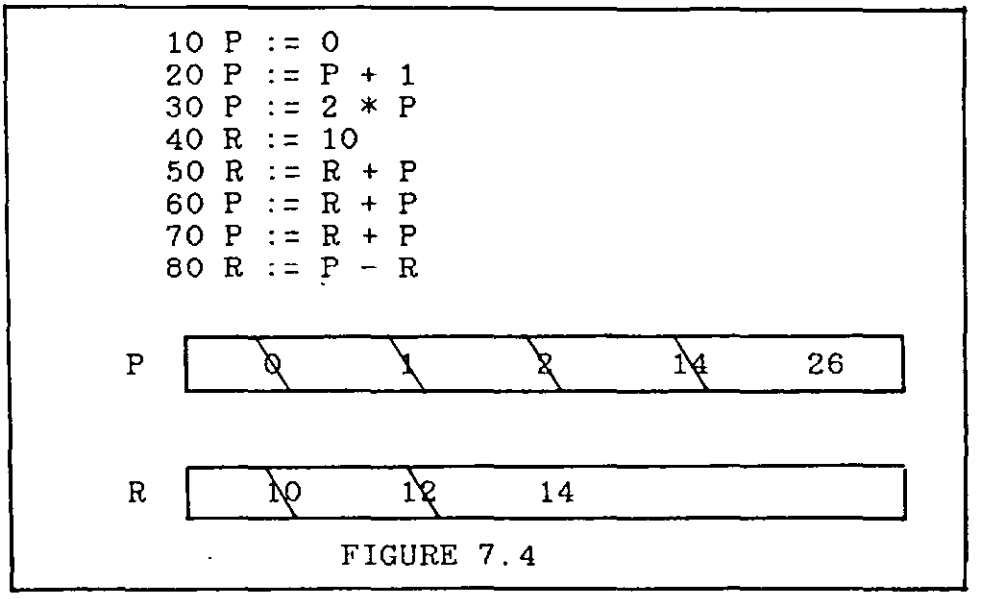


FIGURE 7.4

These problems were all done very well and caused no difficulty; indicating that the meaning of the assignment statement was well understood. None of the examples given at this stage performed any recognisable task, (it is very difficult to devise a useful program with only assignment statements), so the following example was shown before considering the input statement:

```
10 PRINCIPAL := 1000
20 RATE := 12
30 TIME := 3
40 INTEREST := (PRINCIPAL * RATE * TIME) / 100
```

This program fragment suggests the purpose of the assignment statement and was generalised after the introduction of the INPUT statement.

7.4 INPUT AND OUTPUT STATEMENTS

The INPUT statement was introduced by showing a demonstration program which utilised INPUT statements. This program was discussed with the class and then the necessary syntax rules were given. The class was then given numerous problems, which required INPUT statements, to solve and to implement on the computers. This proved to be quite unsatisfactory and many of the students encountered difficulties with the idea of INPUT. In the second year the students were given numerous demonstration programs to run for themselves before the INPUT statement was actually introduced. This was much more satisfactory. These difficulties are discussed in chapter 8.

The use of variables in PRINT statements caused no such difficulties. The approach was that items in quotation marks were literally written on the screen but items without quotation

marks were taken to be variables and so their values were written instead. This was readily accepted.

There are many problems that can be solved with just input, output and assignment statements, many of which are identical in structure. For example:

1. Input the length and width of a rectangle and output its area.
2. Input the radius of a sphere and output its volume.

Many examples of this nature were completed successfully by the students in each year.

7.5 STRUCTURE DIAGRAMS

It was decided to use a number of very familiar, non-computer algorithms to establish the use of diagrams before any computer algorithms were encountered. No formal description of the diagrams' syntax was given at this stage but worksheets were given at each session to be completed during the last five or ten minutes of the class.

The first example given was a description of how to make a phone call (figure 7.5):

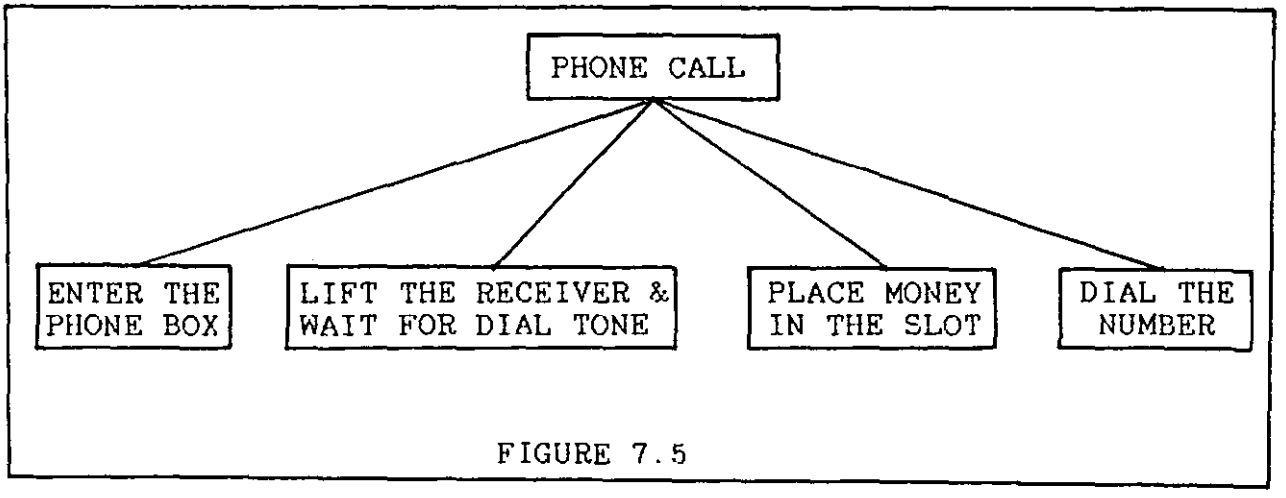


FIGURE 7.5

It was explained that the four boxes in the second row were a description of the statement in the top row and that they should be carried out from left to right to achieve the desired effect. After this minimum of explanation the students were asked to complete similar examples, all concerning familiar activities. Their task was to assign a given list of statements to the appropriate boxes in a diagram. A typical example is shown in figure 7.6:

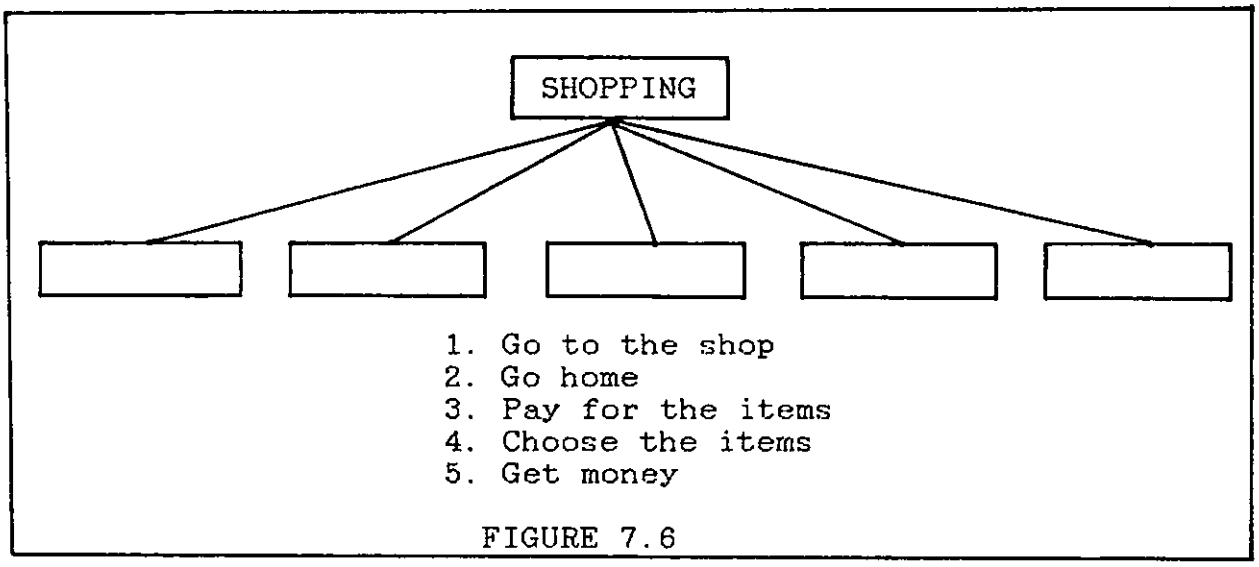


FIGURE 7.6

The first worksheet contained six such problems and was completed, with no errors, by all the students.

Even though these particular algorithms were trivial, they were very useful because they introduced the idea of a structure diagram in a simple manner. They also reinforced the need to get statements into the correct order and to write these sequences of statements horizontally. The next step was to introduce diagrams with two levels and these were also completed easily and correctly by all the students. The final stage in this process was to give partially completed diagrams without a list of the missing statements. In some cases they were

required to fill in only second level statements but in others they were given the second level solution (or most of it) and were asked to fill in a blank in the first level. This latter type of exercise is important as it forces students to think 'procedurally', i.e. it requires them to consider the overall purpose of a group of statements. An example of this type of problem is given in figure 7.7.

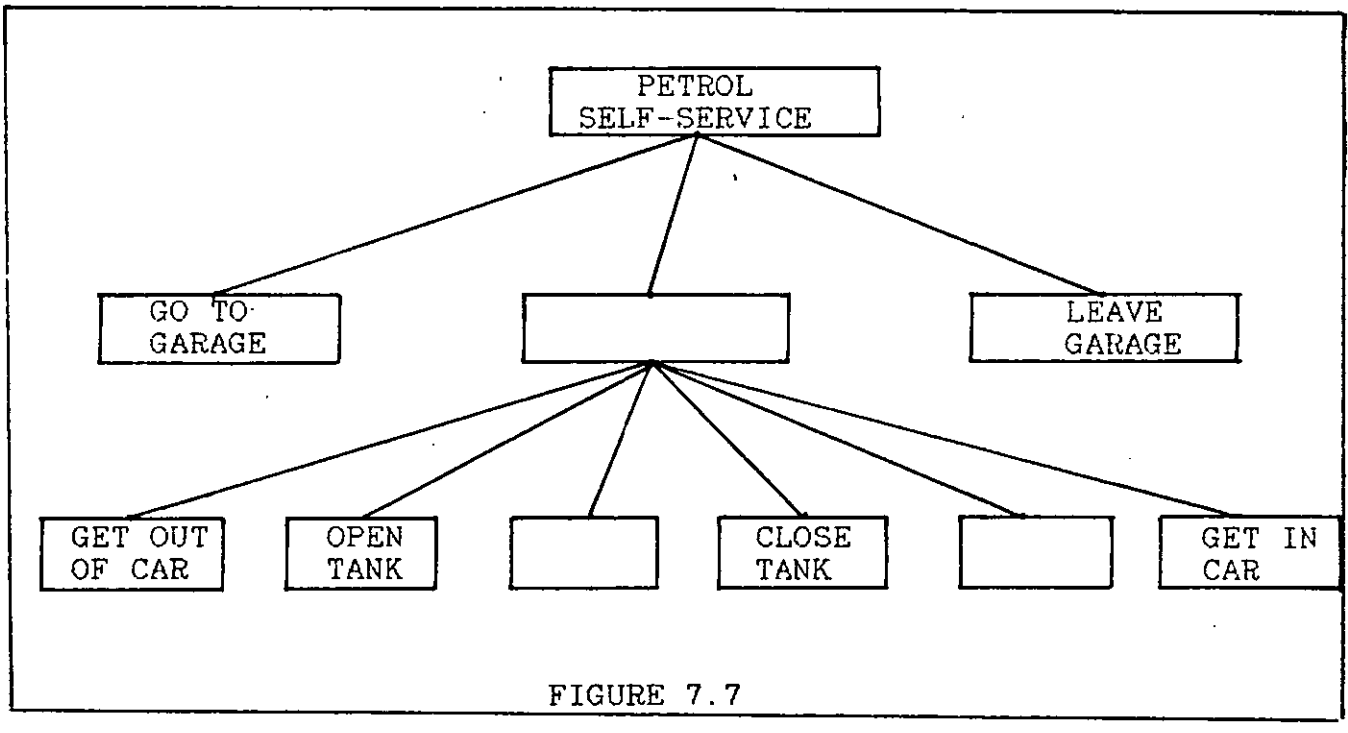


FIGURE 7.7

The missing statements at the second level are 'put petrol in tank' and 'pay for petrol'. All of these second level statements then constitute the action 'buy petrol' and this is the required first level statement.

When all of these worksheets were completed, over a period of two weeks, the topic of Top-Down design was introduced more formally. The first 'computer' problem solved with a diagram

concerned the cost of laying a path around a rectangular garden (figure 7.8). This type of problem was familiar to the students from their work in junior cycle mathematics. The diagram for this problem may seem very complex at first sight but is quite simple if read correctly. The method is to cover all but the top level of the solution with a blank page. A check is then made to confirm that the statements at this level constitute a complete solution to the stated problem. When this is done the blank page is moved down to the next level of the diagram. A check is then made to ensure that the statements at this level are correct refinements of the statements at the first level. This process is repeated until the whole diagram has been read.

7.6 TOP-DOWN METHODOLOGY

In solving problems with structure diagrams the students were encouraged to ask themselves the following questions, based on those suggested by Polya [12], at each level:

1. What am I trying to find?
2. What must I know to find it?
3. a) Am I given what I need to find it?
   b) Can I calculate what I need to find it?

If these questions are consistently applied, the result is always a well organised Top-Down solution. Consider the following example :

Write a program to calculate nett pay if gross pay, tax free allowance (TFA) and rate of tax are input.

The solution to this problem (figure 7.9) may be derived as follows:

1. Trying to find nett pay.

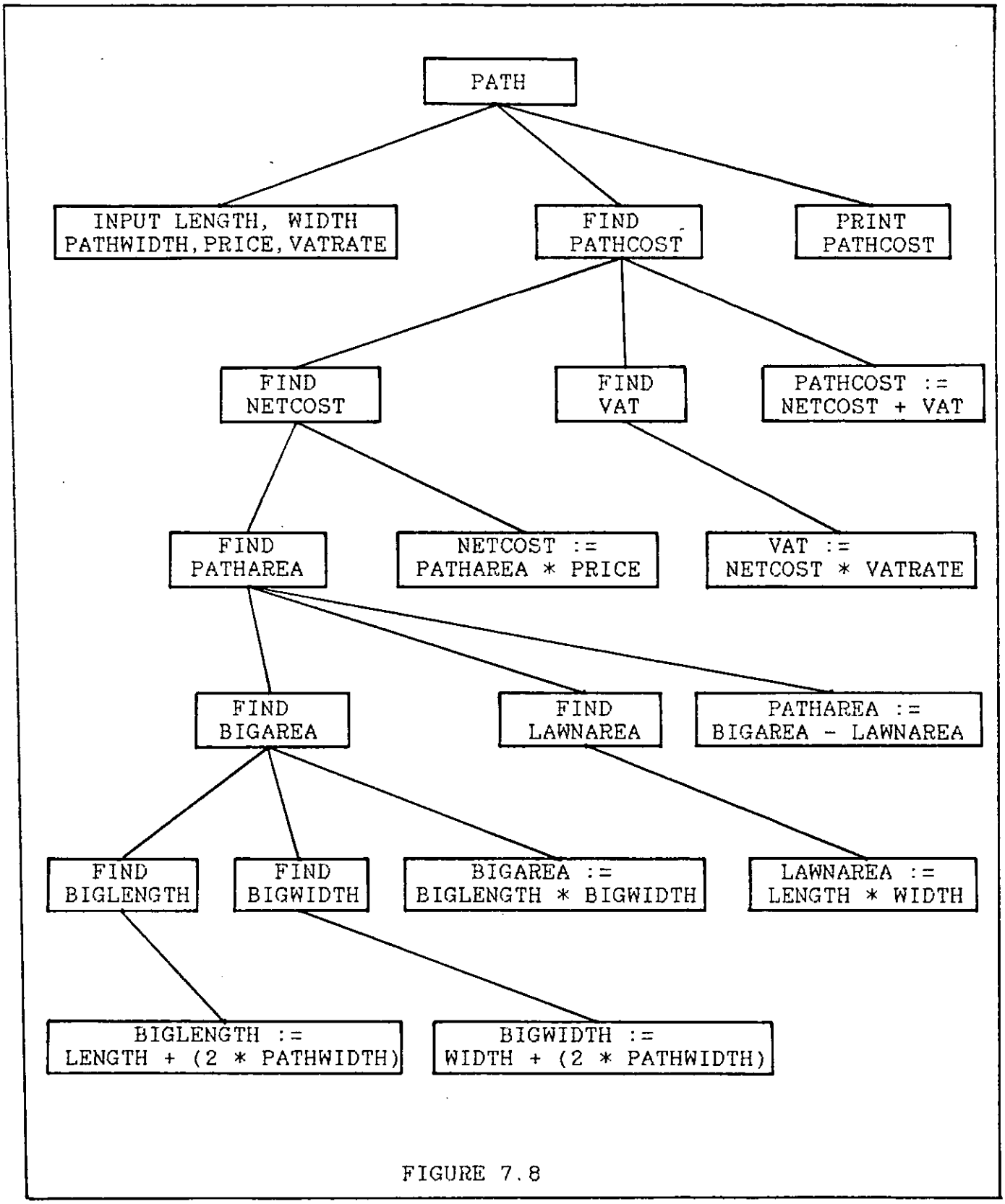

PATH
INPUT LENGTH, WIDTH
PATHWIDTH, PRICE, VATRATE
FIND
PATHCOST
PRINT
PATHCOST
FIND
NETCOST
FIND
VAT
PATHCOST :=
NETCOST + VAT
FIND
PATHAREA
NETCOST :=
PATHAREA * PRICE
VAT :=
NETCOST * VATRATE
FIND
BIGAREA
FIND
LAWNAREA
PATHAREA :=
BIGAREA - LAWNAREA
FIND
BIGLENGTH
FIND
BIGWIDTH
BIGAREA :=
BIGLENGTH * BIGWIDTH
LAWNAREA :=
LENGTH * WIDTH
BIGLENGTH :=
LENGTH + (2 * PATHWIDTH)
BIGWIDTH :=
WIDTH + (2 * PATHWIDTH)

FIGURE 7.8

2. Must know gross pay and tax due.

3. a) Gross pay has been input.

   b) Must find tax due.

Question 1 supplies the first level solution and questions 2 and 3 give the second level solution. Question 3b naturally leads to a repetition of these questions and a refinement of the statement FIND TAXDUE:

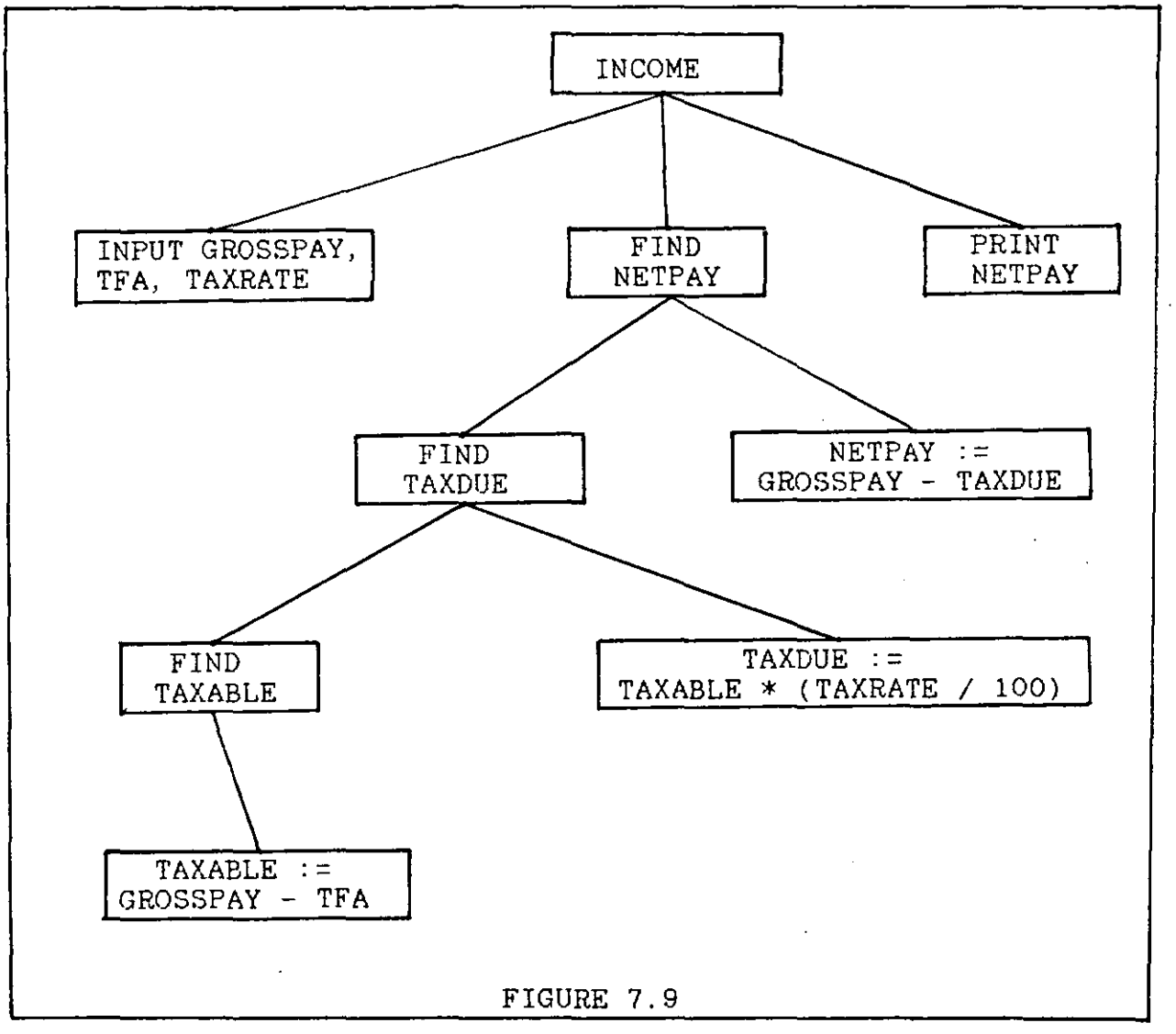


FIGURE 7.9

1. Trying to find tax due.

2. Must know taxable income and tax rate.

3. a) Tax rate has been input.

b) Must find taxable income.

This constitutes the third level solution and again question 3b leads back to question 1 and a refinement of FIND TAXABLE.

1. Trying to find taxable income.
2. Must know gross pay and TFA.
3. a) Gross pay has been input.

   b) TFA has been input.

This is the level 4 solution and completes the analysis.

The programming exercises given with this section were changed, as the ones given in the first year were too difficult (appendix A, p.212). Six new homework problems (appendix A, p.213) were given. These seemed to have been set at the correct level and were answered very well.

7.7 THE CONDITIONAL CONTROL STRUCTURE

It was pointed out that there were many problems that could not be solved by purely 'sequential' algorithms and that many solutions require choices to be made. Printing all the positive numbers from a list of positive and negative numbers was cited as an example. In this case a choice, to print or not to print, must be made for each number in the list. The choice is made by examining a condition, i.e. is the number greater than zero?

The first diagram used (figure 7.10) concerned the problem of printing the greater of two numbers. This diagram was discussed in detail. The boolean conditions controlling access to each branch were described as 'guards' which had to be TRUE before that branch could be executed. The rules for translating

diagrams containing conditional statements into COMAL were given and this program fragment was then coded.

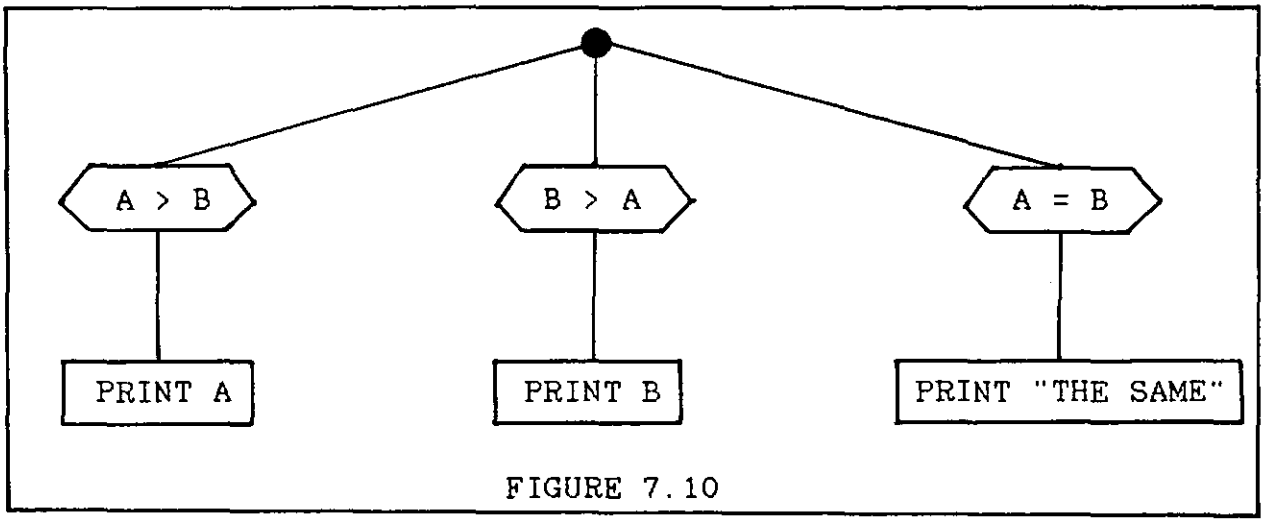


FIGURE 7.10

7.7.1 RULES FOR TRANSLATION OF IF STATEMENT INTO COMAL:

1. "Walk" around the structure in the same way as described in chapter 4.
2. On the first encounter with the ● symbol write the word IF, followed by the first condition, followed by the word THEN (all on one program line).
3. Write the statement(s) that are guarded by the first condition on the line(s) immediately after it.
4. On subsequent encounters with the ● symbol write the word ELIF, followed by the next condition, followed by the word THEN (all on one line).
5. Write the statement(s) that are guarded by this condition on the line(s) immediately after it.
6. On the final encounter with the ● symbol write the word ENDIF (on a program line of its own).

The COMAL fragment derived from the diagram in figure 7.10 is therefore as follows:

```
100 IF A > B THEN
110    PRINT A
120 ELIF B > A THEN
130    PRINT B
140 ELIF A = B THEN
150    PRINT "THE SAME"
160 ENDIF
```

The students were then given several simple problems, requiring conditional statements, for homework. In each case they were required to draw a diagram and write a COMAL program.

7.7.2 BOOLEAN OPERATORS

To introduce boolean operators, each student was given a sheet containing information, examples, truth tables and problems. Each operator was introduced by a simple non-computer example as follows:

IF IT IS FINE AND THE POOL IS OPEN, JOHN WILL GO SWIMMING.

IT IS FINE	THE POOL IS OPEN	DOES HE GO SWIMMING?
TRUE	TRUE	YES
TRUE	FALSE	NO
FALSE	TRUE	NO
FALSE	FALSE	NO

This table was filled in with the help of the class and then a more formal truth table was drawn up:

Exp.1	Exp.2	Exp.1 AND Exp.2
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
FALSE	TRUE	FALSE
FALSE	FALSE	FALSE

Then, again with the help of the class, a general statement about the AND operator was drawn up:

Exp.1 AND Exp.2 is TRUE

only when Exp.1 is TRUE and Exp.2 is TRUE

Similar examples were used to introduce the OR and NOT operators.

The next stage was to present 'real' boolean expressions, which they were required to evaluate for given values of the variables involved, as in figure 7.11. Numerous examples of this type were done in class. Initially, the examples involved only one operator but later examples involved combinations of two or three operators. The students were very accurate when evaluating these expressions and seemed to enjoy working on them. A homework sheet containing about fifty such examples involving AND, OR, and NOT was assigned and they were asked to do ten particular examples from this. Most of the class did all the problems on the sheet and most had all the answers correct.

	X = 1 Y = 1	2 1	1 2	2 2	1 3
X > 1 AND Y > 1					
X > 1 AND Y < 2					
X = 2 AND Y = 2					
X < Y AND Y = 2					
X > Y AND Y <> 1					

FIGURE 7.11

When evaluating complex boolean expressions, the students were discouraged from trying to comprehend the totality of the

expression but were shown how to approach the task of evaluation in a systematic way. This involved writing the value (T or F) of each part beneath the expression and then using the previously defined rules for combining these values. This may be illustrated by the following example:

Evaluate the following expression for X = 2.

```
  (X < 10   OR  X > 20)  AND  (X MOD 2 = 0  AND X MOD 3 <> 0)
= (  T      OR    F   )  AND  (      T      AND      T     )
=             T            AND                  T
=                          T
```

Up to this the emphasis had been on evaluating given expressions and this had been learnt very well by all the students. However, the really important skill in programming is to construct boolean expressions. This was approached by using a COMAL program which simulated a game in which two dice are thrown. A win was defined as either:

a. The two dice are the same.

or b. The sum of the dice exceeds nine.

```
 10 // DICEGAME SIMULATION
 20 //
 30 ////////////////////////////////////////////
 40 CLEAR
 50 RANDOM
 60 DIE1 := RND(1,6)
 70 DIE2 := RND(1,6)
 80 SUM := DIE1 + DIE2
 90 ////////////////////////////////////////////
100 PRINT "SCORE ON FIRST THROW IS  .....";DIE1
110 PRINT
120 PRINT "SCORE ON SECOND THROW IS .....";DIE2
130 ////////////////////////////////////////////
140 // NOW MAKE DECISION ABOUT THE PRIZE
150 //
160 IF SUM > 9 OR DIE1 = DIE 2 THEN
170    CURSOR 1,20
180    PRINT "YOU HAVE WON A PRIZE !!!!"
190 ENDIF
200 ////////////////////////////////////////////
210 END
```

The program was discussed in class to ensure that everyone understood how it worked. The students were then asked to alter the program to simulate ten new games in which the definition of a win was changed as follows:

1. Sum exceeds 7 or both dice are twos.
2. Both dice are even.
3. Both dice are odd.
4. Both dice are bigger than 4.
5. At least one die is bigger than 4.
6. DIE1 is bigger than 3 and DIE2 is less than 3.
7. One die is bigger than 3 and the other one is smaller than 3.
8. Both dice are equal and both are less than 5.
9. Neither die is smaller than 3.
10. The sum does not exceed 8 and neither die is less than 3.

This type of exercise was useful as it allowed the students to focus exclusively on the boolean expression without having to consider the other parts of the program. These exercises were done very well with an average of 80% accuracy. Many of the errors were syntax errors. In particular, they were inclined to write A = B = C instead of A = B AND B = C but this type of error, which is detected by the COMAL interpreter, did not persist after they had implemented a few such programs on the machines.

7.8 THE ITERATIVE CONTROL STRUCTURE

It was pointed out that many problems required the same basic steps to be performed on different sets of data and that it would be very wasteful to write the same code for each data set. The example of printing the squares of the first twenty numbers was cited. It was suggested that it would be possible to write a separate routine for each of the numbers but that this would be out of the question if the first two thousand numbers had to be processed. The problem could be solved, however, by allowing a section of the program to be executed twenty times, processing a different number each time. The diagram in figure 7.12 was then shown and discussed. The round box was described as the 'loop control box' and the boolean condition inside it was called the 'loop guard'. The meaning of the diagram was discussed at length and the similarities between the iterative and the conditional statements were stressed (the chief difference is that the condition is tested again after the guarded statements have been executed). It was emphasised that all variables in the loop guard must be initialised before the condition is tested (hence the need to set COUNT = 1 initially). It was also emphasised that the programmer must ensure that the condition eventually becomes FALSE to avoid the loop being executed infinitely.

7.8.1 RULES FOR TRANSLATING ITERATIVE STRUCTURE INTO COMAL

1. On the first encounter with the loop control box write the word WHILE, followed by the loop guard, followed by the word DO (all on one program line).

2. Write the statements that are guarded by the loop guard.

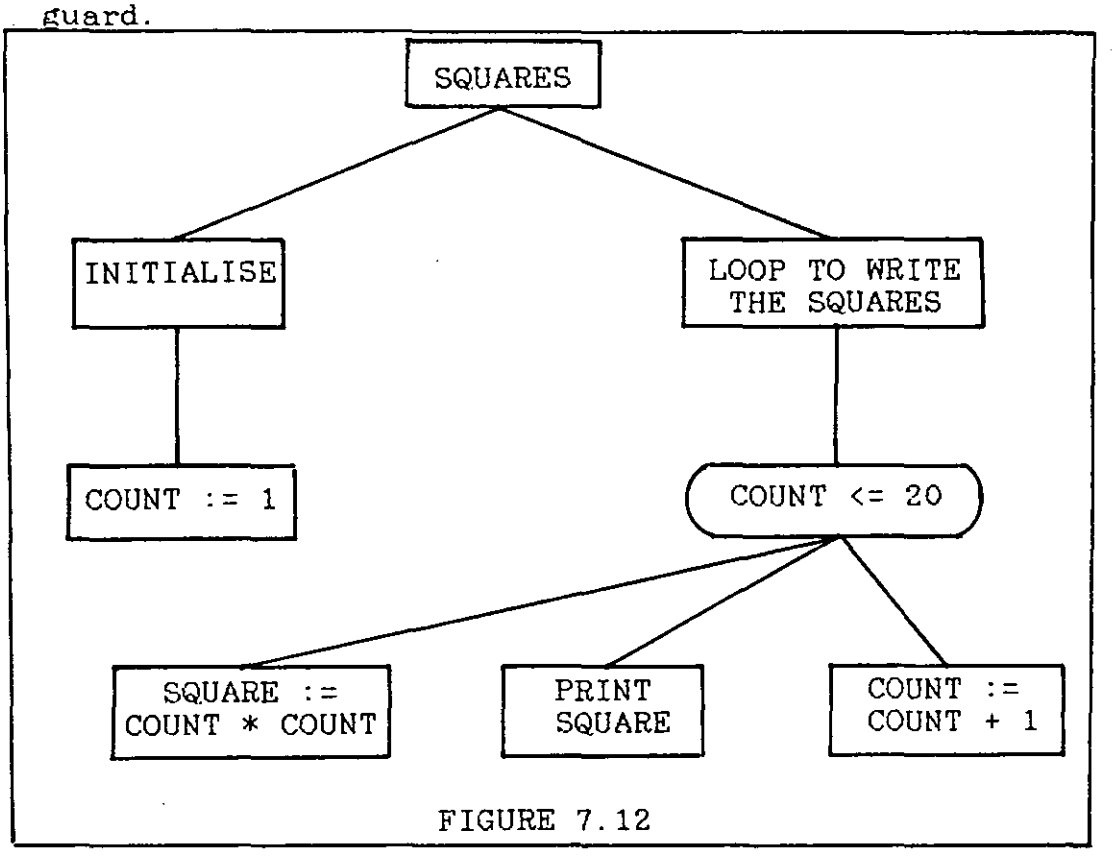


FIGURE 7.12

3. On the second and final encounter with the loop control box write the word ENDWHILE (on a separate program line).

The COMAL fragment derived from figure 7.12 using these rules is as follows:

```
100 COUNT := 1
110 WHILE COUNT <= 20 DO
120   SQUARE := COUNT * COUNT
130   PRINT SQUARE
140   COUNT := COUNT + 1
150 ENDWHILE
```

Similar problems were then solved by the students, using structure diagrams, in class. These were then coded in COMAL for homework and implemented on the machines without any difficulty.

7.8.2 FIXED ITERATION VS. INDEFINITE ITERATION

These problems illustrated a very important routine which occurs frequently in later problems, i.e. to execute a loop a fixed number of times (fixed iteration). Using a WHILE loop this may be done in a number of different ways. In the program on the left, the counter variable is given a value of one before the loop is entered, while in the version on the right it is initialised to zero.

```
100 COUNT := 1                          100 COUNT := 0
110 WHILE COUNT <= N    DO              110 WHILE COUNT < N    DO
 '                                       '
 '   (guarded statements)                '   (guarded statements)
 '                                       '
190    COUNT := COUNT + 1               190    COUNT := COUNT + 1
200 ENDWHILE                            200 ENDWHILE
```

The outcome is the same in each case because the loop guards are different. It was felt that it was important to choose one method and to use it consistently in order to avoid confusion. The former method was chosen (even though the other one is used in many textbooks) because:

1. It seemed to be more in accord with common sense:

   COUNT is 1 on the first iteration.

   COUNT is 2 on the second iteration. etc.

2. It is very often necessary to use the value of COUNT within the loop (as in the SQUARES problem above). The second method would involve some 'fixing' under these circumstances.

It was decided, having established the notion of fixed iteration, to introduce some indefinite iteration problems immediately, lest it might be assumed that all loops must be fixed iteration. The first such problem was:

Write a program which keeps accepting pairs of numbers and printing the larger of each pair. The program should terminate when two equal numbers are input.

The students were first asked to decide if this problem contained a loop. When this was agreed they were asked to decide how many times the loop should be executed. This caused bewilderment because it was impossible to say. They were then asked to state under what circumstances the iteration should stop. This was very easy to state, i.e. when both numbers are equal. It was pointed out that if this was the condition for terminating the loop, then the loop guard must be the opposite of this, i.e. when both numbers are different. They were then asked to say what should be done inside the loop and most were able to see immediately that the two numbers should be compared and the larger one printed. The fact that another two values should be input was overlooked by many and even when the complete solution (figure 7.13) was shown there was still some confusion on this point.

In this case and in many others 'running' the solution on the board was found to be helpful. Students were encouraged to 'run' all their solutions in this way as a check on the algorithm before implementing it on a machine. The technique used was as shown in figure 7.14.

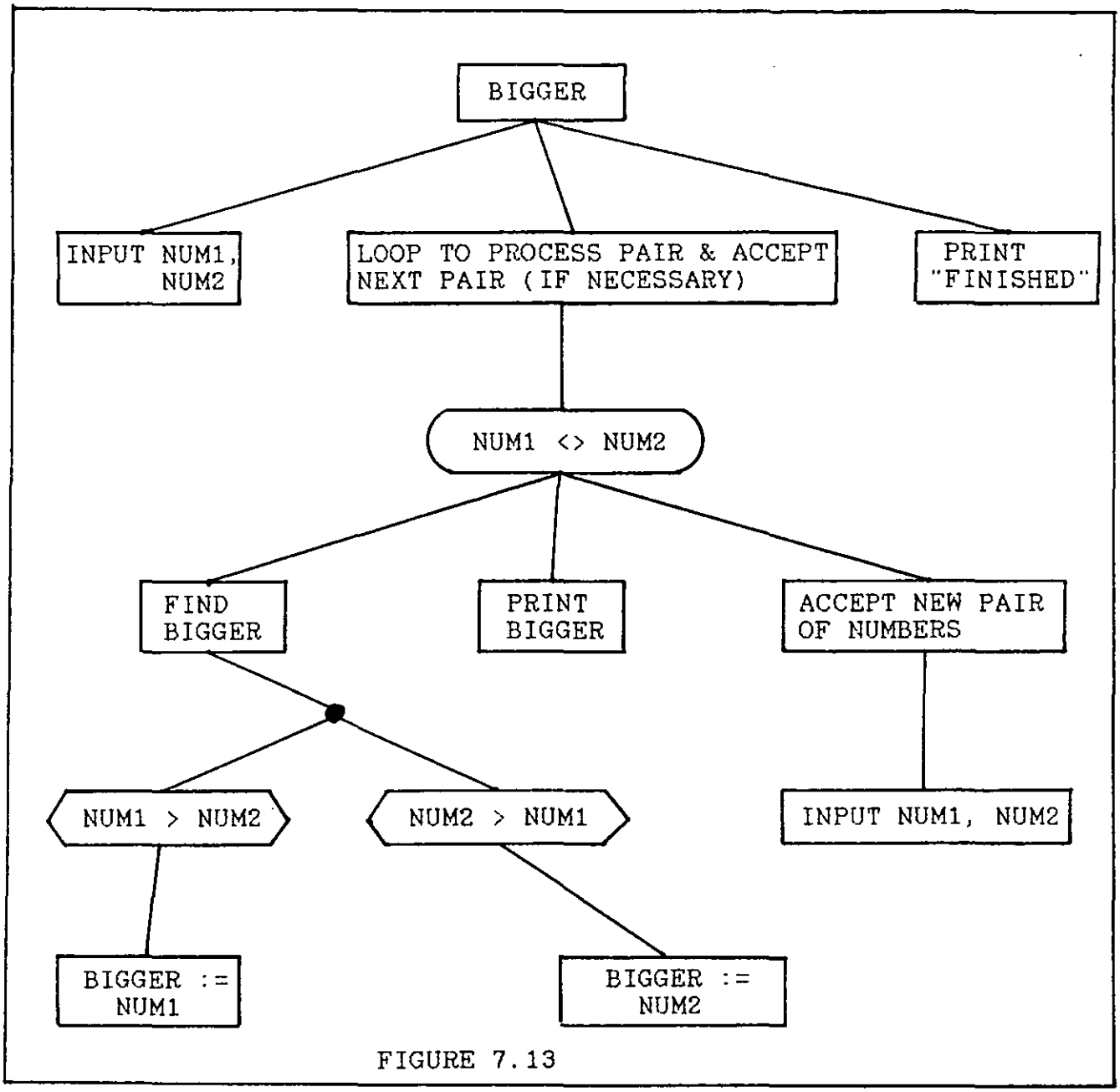


FIGURE 7.13

VALUES TO BE INPUT	EXPECTED OUTPUT
3 , 2	3
5 , 8	8
7 , 7	"FINISHED"

NUM1	NUM2	LOOP GUARD	OUTPUT
3	2	TRUE	3
5	8	TRUE	8
7	7	FALSE	"FINISHED"

FIGURE 7.14

The values to be input are decided in advance along with the expected output. A trace is then kept of the values of all the variables involved and of the loop guard. It is felt that this is a vital skill and that even though students are often reluctant to apply it that it is worth forcing them to do so (by homework assignments, worksheets etc.). This technique was used during class time, by the teacher, in the hope that the students would see the benefit of such an approach for themselves and apply it in their own work. It was found, however, that students did little that was not specifically required of them. For this reason it is planned to devise worksheets containing programs with loops, along with charts like that in figure 7.14, which the students will be required to complete.

7.8.3 GENERAL METHOD FOR CONSTRUCTING LOOPS

In general, when writing a loop the students were encouraged to ask themselves the following questions:

1. What variables are needed?
2. Is it a fixed iteration or an indefinite iteration loop?
3. What should the loop guard be?
4. What needs to be done inside the loop?
5. What needs to be initialised?
6. How can it be guaranteed that the loop will terminate correctly?

7.8.4 LOOPS TO ADD NUMBERS

The general method may be illustrated by considering the following problem:

Write a program to calculate the number of terms of the series 1 + 3 + 5 + 7 ..... that must be added to give a sum greater than any specified (input) value.

If the six questions outlined above are considered the following answers may be found:

1. Need variables for:
   a) the current total .....SUM
   b) the number of terms added to date ..NUMTERMS
   c) the current term being added ..CURRTERM
   d) the specified limiting value .....LIMIT
2. This is indefinite iteration as it is not known in advance how many times the loop should be executed.
3. The loop should terminate when the value of SUM exceeds the value of LIMIT. This means that the loop guard should be the opposite of this:
   i.e. SUM <= LIMIT
4. Need to add the current term on to the total, generate the next term of the series and increment the number of terms.
5. The loop guard involves both SUM and LIMIT so these must be initialised, SUM is set to 0 and LIMIT is input. It is also necessary to set NUMTERMS to 0 and CURRTERM to 1 as both of these are incremented inside the loop.
6. As SUM is increased each time the loop is executed, its value will eventually exceed that of LIMIT which is not changed inside the loop.

After considering these questions the solution in figure 7.15 was arrived at:

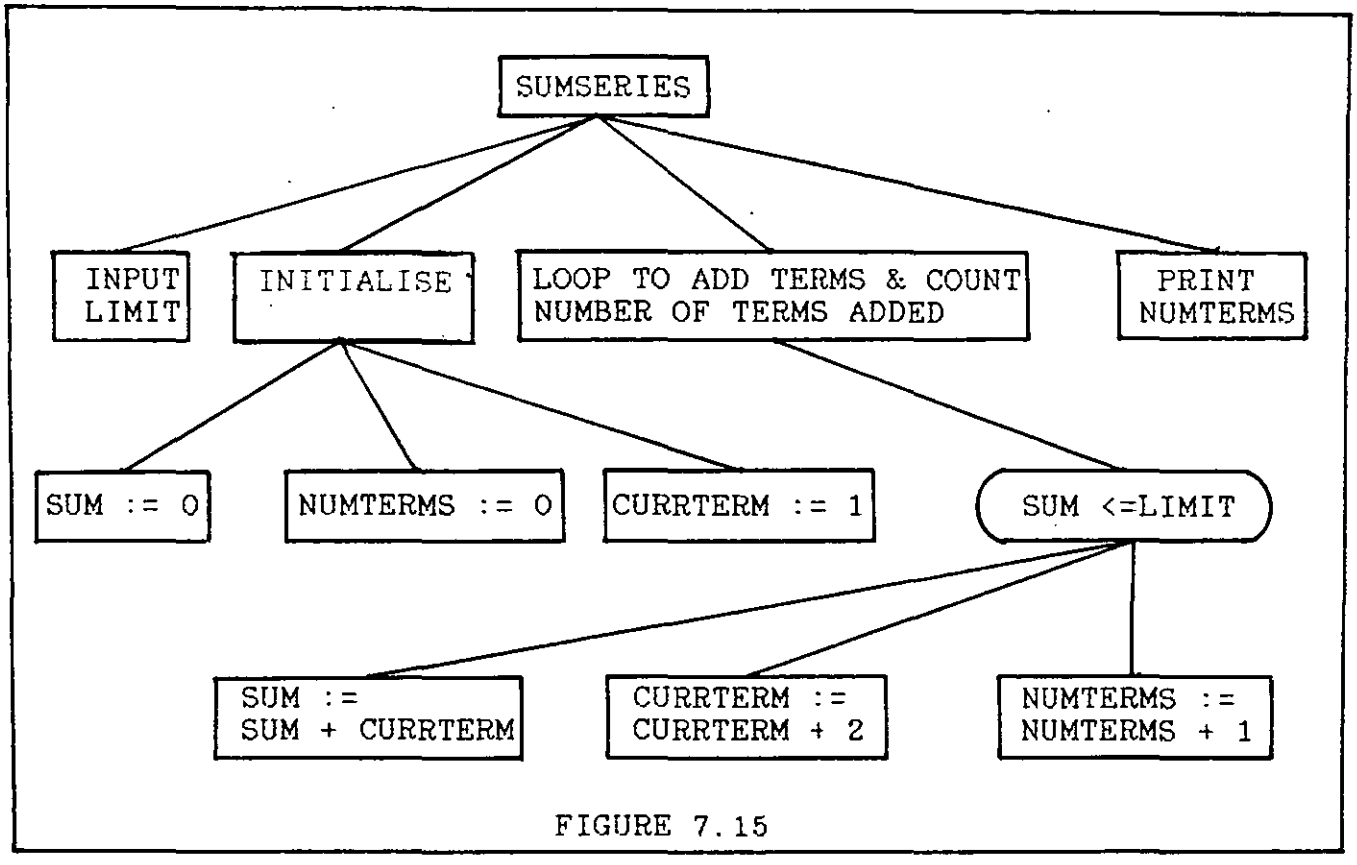


FIGURE 7.15

Several other problems set at this level of difficulty were then solved in class and as homework assignments. In general, these were well done although there were some silly mistakes such as neglecting to initialise or increment variables. These would not have occurred if the procedure outlined above had been carried out in all cases. It was also obvious from the nature of the errors that some students were not 'running' the solutions on paper as outlined in figure 7.14. If all students could be convinced that asking the six questions above before designing the loop, and 'running' the finished product on paper were always worthwhile, then there would be fewer careless errors in their work. As with some other techniques, the good example of the teacher has not been totally effective in these matters. Specific exercises on these techniques will be designed for future courses.

An important point regarding the design of the loop guard for indefinite iteration loops may be seen in the previous example. In difficult cases, it is often simpler to decide on the condition for terminating the loop and then to reverse this using de Morgan's law, than to consider the condition for entry to the loop directly in the first instance. This is particularly useful when the loop guard contains boolean operators. For example, consider a program to simulate throwing a die until a four turns up or the sum of all the throws exceeds thirty. In this case, the condition under which the loop should terminate is obviously:

```
DIE = 4  OR  SUM > 30
```

Reversing this gives the condition under which the loop should be executed, i.e. the loop guard:

```
DIE <> 4 AND SUM <= 30
```

This technique has been found to be most useful in more complex cases but may of course be applied in any situation.

7.8.5 PROBLEMS INVOLVING READ/DATA STATEMENTS

Numerous problems were used which involved READ/DATA statements. These statements had not been introduced up to now because it was felt that the input statement alone was perfectly adequate for programs without loops. It was now felt that a number of important ideas, which would normally be introduced with arrays and files, could be dealt with more simply by using read/data statements. Data which is held in an array or a file is not 'visible' and cannot therefore be thought about as easily as that which is contained in a program listing. However, the algorithms for handling data are much the same, so that anything

learned from these problems should be useful when arrays and files are encountered later.

COMAL contains an end-of-data flag, EOD. This is a system variable, of type BOOLEAN, which only becomes TRUE when all the data in a program has been read. This can be used to access all the data in a program as follows:

```
100 WHILE NOT EOD DO
110   READ N
  .
  .
200 ENDWHILE
```

It was decided not to use this facility because:

1. BOOLEAN variables had not been introduced.
2. The construction of 'do-it-yourself' end-of-data flags helps to focus attention on the need for good organisation of data in a program.
3. EOD is not generally available in other languages.

When designing solutions to problems involving read/data statements, the practice of writing samples of the data (including the end-of-data flag where appropriate) beside the diagram was adopted. This was to emphasise the importance of data organisation and also because the diagram would be meaningless without this information. Throughout these problems the data has been organised in 'logical units' rather than to save memory or to speed implementation. For example, if the data were to contain names and ages it would be organised as follows:

```
DATA "JOE",14
DATA "MARY",15
```

rather than the more usual:

```
DATA "JOE",14,"MARY",15
```

This latter organisation is very difficult to read and to debug when there is even a modest amount of data involved.

It was further decided when constructing end-of-data flags to use sufficient terminal values to match the overall organisation of the data, as in the example on the left, rather than the normal practice of just using one flag , as on the right.

```
100 DATA "JOE",14
110 DATA "MARY",15
 ,
 ,
200 DATA "END",-1
```

```
100 DATA "JOE",14
110 DATA "MARY",15
 ,
 ,
200 DATA "END"
```

This organisation allows the use of a clearer algorithm (on the left) to process the data, as opposed to the more usual method (on the right).

```
10 READ NAME$
20 READ AGE
30 WHILE NAME$ <> "END" DO
40          ,
50          ,
60   READ NAME$
70   READ AGE
80 ENDWHILE
```

```
10 READ NAME$
20 WHILE NAME$ <> "END" DO
30   READ AGE
40      ,
50      ,
60      ,
70   READ NAME$
80 ENDWHILE
```

In the program on the left, the two values being read from the same data line are read together. In the other program, this is not possible and one value is read at the start of the loop and the other is read at the end. There is no doubt that the former program is much clearer and simpler. This can only be achieved if the data statements are organised as outlined above.

The syntax and semantics of the read/data construct were explained using many simple examples. It was emphasised that the programmer must ensure that string values and numeric values are read into variables of the appropriate type. The concept of a data pointer was introduced and a little arrow was drawn on

the board to denote the pointer. This was moved forward as each item was read. The RESTORE statement was also introduced along with its COMAL variations. These allow the pointer to be restored to a particular line-number or to a LABEL.

All of the early problems used in this section involved algorithms which were already quite familiar (adding numbers, finding averages etc.), the only difference being that the data were read rather than input. This served the purpose of familiarising the students with the read/data construct without introducing any very difficult problems.

7.8.6 MORE DIFFICULT LOOPING PROBLEMS

The next, more difficult, group of problems concerned linear searches. This is an important algorithm and requires careful consideration. The first and simplest of these problems was:

> Write a program to say whether or not a name, input at the keyboard, is contained in the data list. The end of the data is marked by the 'name' "END".

The loop for this problem was designed as outlined in 7.8.3. As the loop can be terminated for either of two different reasons it is necessary to include a conditional statement after the loop to find out which part of the loop guard became FALSE. A complete solution is shown in figure 7.16. Many problems which were very similar to this, also involving linear searches but with added complexities, were then solved both in class and for homework and these were done exceptionally well.

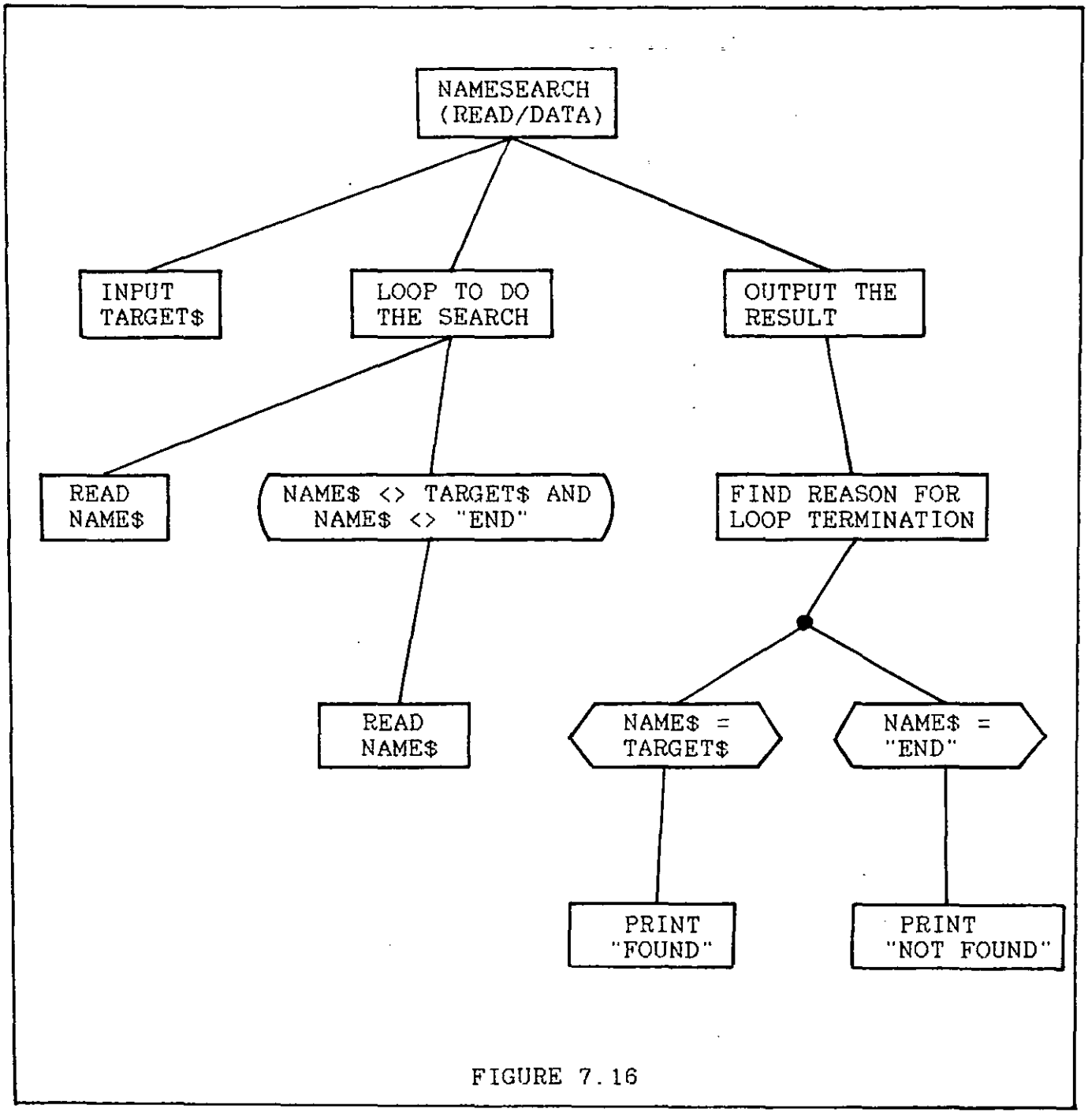

NAMESEARCH
(READ/DATA)
INPUT
TARGET$
LOOP TO DO
THE SEARCH
OUTPUT THE
RESULT
READ
NAME$
NAME$ <> TARGET$ AND
NAME$ <> "END"
FIND REASON FOR
LOOP TERMINATION
READ
NAME$
NAME$ =
TARGET$
NAME$ =
"END"
PRINT
"FOUND"
PRINT
"NOT FOUND"

FIGURE 7.16

7.9 PROCEDURES

It was decided to use problems which were fairly simple, and which only used algorithms which were already quite familiar, in the initial stages of this section. The problems, though simple, were much bigger than any which had been previously encountered. This arrangement allowed the students to concentrate on the overall organisation of the solution, without having to consider the minute details of each sub-algorithm.

The first problem used was:

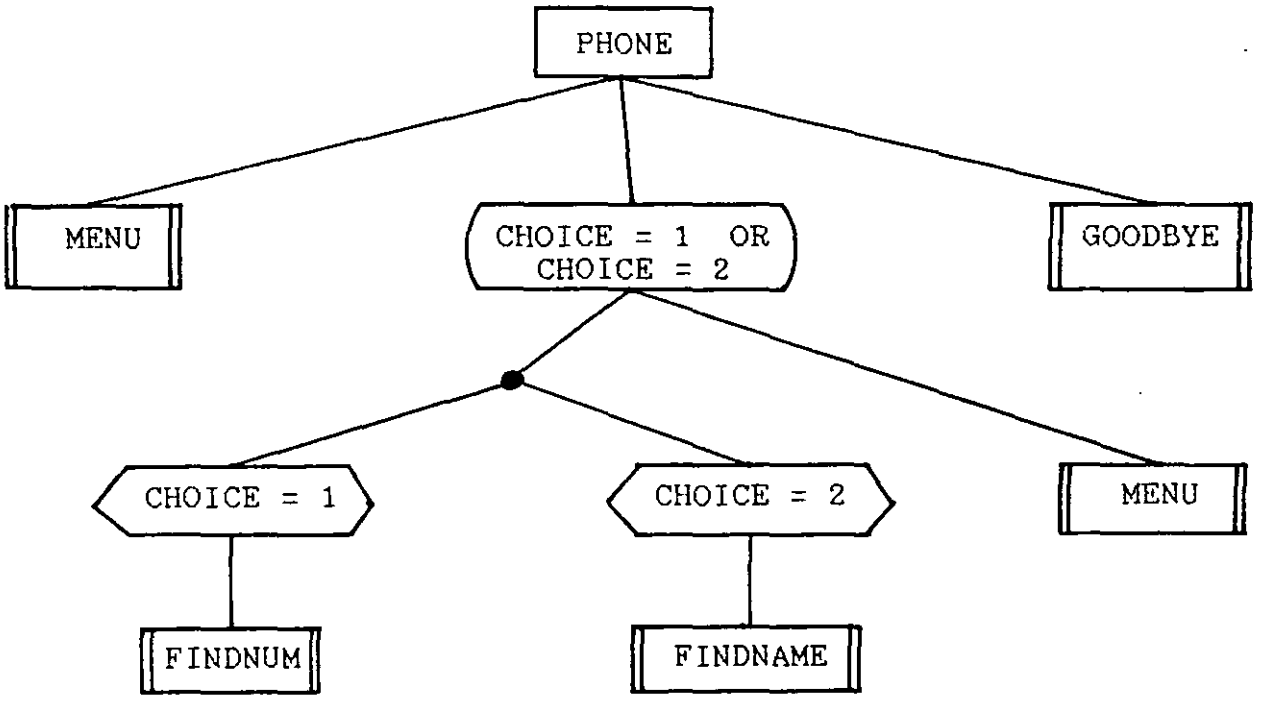
> Write a program which allows the user to either a) input a person's name and find the appropriate phone number or b) to input a number and find the name. The names and telephone numbers are organised in data statements as follows:
>
> ```
> DATA "MARY", 234156
> DATA "ANNE", 763452
> ```
>
> The end of the data is marked by:
>
> ```
> DATA "END", 0
> ```
>
> If the input name or number is not in the data then the program should output a suitable message. When the name or number has been supplied the user should again be given the option of inputting another name or number or of quitting the program.

This problem was approached in a different way to the previous problems in the course. The problem definition was not given to the students but a well structured, menu-driven program to solve the problem was supplied on disk instead (FONEDEMO.CSB). This program is listed in appendix B, p.220 The students were asked to run the program without being told what it was about or how

it was structured. After each student had used the program for about five minutes a class discussion was held during which the students were asked to describe what the program did. They did this very well, discussing the program at a very high level without concerning themselves with how the searches were carried out. The consensus of what the program did was:

1. The user is given three options:
   A. Find a phone number given a name.
   B. Find a name given a phone number.
   C. Quit the program.
2. If the user chooses A or B then the required information is requested and the program carries out a search.
3. After giving the result of the search, the program returns to the first section and the user is again given the three options.
4. The program terminates when, and only when, the user chooses the QUIT option.

The remarkable thing about this description is that it exactly mirrors the structure of the main program (figure 7.17). The double sided boxes were called PROCEDURES and were simply described as 'sections of the program, each of which performed a particular task'. It was emphasised that once a procedure was used in a structure diagram that it was essential to specify its purpose. This specification, which must be shown with the diagram containing the procedure, should later be used as a comment (REM statement) in the program listing.



MENU	: Displays the options and requests the user to input a value to a variable called CHOICE.
FINDNUM	: Asks for a name and then searches for the appropriate phone number.
FINDNAME	: Asks for a phone number and then searches for the appropriate name.
GOODBYE	: Indicates to the user that the program has terminated.

FIGURE 7.17

After the diagram had been discussed in detail the COMAL code for the main program was written. This introduced the COMAL keyword EXEC which is used to call a procedure.

```
220 EXEC MENU
230 WHILE CHOICE = 1 OR CHOICE = 2 DO
240    IF CHOICE = 1 THEN
250       EXEC FINDNUM
260    ELIF CHOICE = 2 THEN
270       EXEC FINDNAME
280    ENDIF
290    EXEC MENU
300 ENDWHILE
310 EXEC GOODBYE
320 END
```

This method of introducing procedures was found to be very successful. It emphasised the notion of a procedure as a block within a program, which is designed to carry out some specific, well-defined task. This is a very powerful concept and is missed by many textbooks which introduce procedures as 'containers' for chunks of code sandwiched between PROC and ENDPROC statements. It also suggests that a procedure may be used as long as the programmer knows exactly WHAT it does, even if its internal structure is not known or understood in detail. In this sense, it prepared the way for the introduction and use of library procedures.

7.9.1 CODING STYLE

After the COMAL statements for calling (EXEC) and defining (PROC/ENDPROC) procedures had been discussed in detail, a full listing of the FONEDEMO program was given to each student. The following important points of coding style were stressed:

1. The main program should come first followed by the procedure definitions.
2. The main program should be short and should consist mostly of procedure calls.
3. Each procedure should contain a one sentence REM statement describing what it does. It was suggested that if this could not be done in one short sentence then further refinement should be considered.
4. The program listing should be easy to read. 'Empty' REM statements should be put at the beginning and the end of each section.

5. All DATA statements should be placed together, preferably at the end of the listing.
6. All initialisation should be done at the beginning of the listing.
7. Every program should contain the name of the author, the date it was written and a brief description of what it does, at the beginning of the listing.

It should be emphasised that the motivation for these guidelines was based on making the students' programs clearly readable and understandable and that issues such as speed of execution and efficient use of memory were not considered to be relevant.

The students were then asked to write a menu-driven program which gave the user the choice of calculating the area of a rectangle, the area of a circle, the volume of a cylinder or the volume of a cone, (or of quitting). They were told to model their programs on the previous example. This was done very well and demonstrated that it is not too difficult to write a long program if a good organisational framework is worked out in advance. The coding guidelines above were well adhered to and the overall structure, in most cases, was identical to that in the phone problem. An example of this work, MENUEXER.CSB, is supplied on disk. The only deviation from the proposed structure was that some students called the MENU procedure at the end of each of the 'calculating' procedures, rather than just once at the end of the loop in the main program. This worked correctly but made the listing more difficult to understand. This was because it was not obvious from reading

the main program that the value of CHOICE was being changed each time the loop was executed and therefore it was not obvious that the loop would terminate properly. When this was pointed out to the students concerned, they were reluctant to accept that it was important enough to warrant changing their programs but were eventually persuaded to do so.

7.9.2 LIBRARY PROCEDURES

Some time was then spent on the use of library procedures and the COMAL commands LIST and ENTER were introduced. The use of the command LIST for two completely different purposes in COMAL is one of the few design faults in the system. When LIST is followed by a filename it writes the contents of memory to disk, under that filename, as a string of ASCII characters and appends the suffix .CML to the filename. The file can be retrieved without disturbing the contents of memory by using the command ENTER followed by the filename. Thus, if there is no clash of line-numbers, this can be used to store and retrieve procedures which may be used in many programs. It is normal to use very big line-numbers when designing such procedures, as this minimises the likelihood of a clash of line-numbers when the procedure is added into memory. A few useful procedures to enhance screen displays (INVERSE, NORMAL and FLASH), one to sound the Apple's bell (BELL), and one to halt program execution until the spacebar is hit (SPACEBAR), were supplied. The students were shown how to LIST and ENTER these and how to utilise them in their programs. They were very amused by these and, for some time after, included them at every conceivable opportunity in their programs. All of these procedures

contained statements that had not been introduced in class (PEEK and POKE) and so the point was made that it was not always necessary to understand the minute details of a procedure in order to utilise it effectively.

Another large program (QUIZDEMO.CSB) was supplied on disk for the students to use and discuss in the same way as FONEDEMO.CSB. A listing of the program is in appendix B, p.223. This program conducted a multiple choice quiz and utilised all the procedures mentioned above. Although it merely asks three questions, the overall structure is very general and could be adapted to ask any number of questions, on any topic, whether they were stored in data statements or in a separate diskfile. The program was used and discussed at length in the same way as FONEDEMO. They were then asked to design a similar program to conduct a quiz on chemical names and symbols. There was an added complication in this assignment, as the user was to be given the option of matching chemical names with chemical symbols, or vice versa. (Those not studying chemistry were given the option of writing a similar program on any topic of their choice). To do this, it was necessary for the programmer to combine the features of the two demonstration programs above, as the program had to implement a multiple choice quiz AND be menu-driven.

This problem demanded the construction of two different sets of data. One set was needed for the option where the user was to be tested on chemical names. In this set, each data line had to contain a chemical symbol, three possible answers and the response (a letter) associated with the correct answer,

e.g. DATA "C", "COPPER", "CARBON", "CHLORINE", "B"

For the other set of data, where the user was to be tested on chemical symbols, each line had to contain a chemical name, three possible symbols and the letter associated with the correct response,

e.g. DATA "SODIUM", "S", "So", "Na", "C"

While two sets of data were required, it was possible to manipulate them both with the same procedure and this is how it was done by all the students. In order to get the data pointer to the correct set of data the RESTORE LABEL statement was used by all the students, except one. This girl was unaware of the existence of this feature but still got her program to work correctly by placing 'flags' in the data. This led to a slightly more cumbersome but impressive program (QUIZEX2.CSB). Other examples of the students' work on this problem are also on disk (QUIZEXER.CSB and CHEMQUIZ.CSB).

7.10 GENERAL APPROACH TO LARGE PROBLEMS

Up to this, the problems dealt with had involved the organisation of various simple and familiar algorithms in large programs. An attempt was now made to demonstrate a method of finding a complete solution to any large problem which might be both unfamiliar and difficult. The problem specification was very long and detailed (figure 7.18) and some time was spent making sure that everybody understood what the program was supposed to do. This was done by asking what should appear on the screen at each stage of the program. As this was a menu-driven program the outer structure was already quite familiar and this is shown in figure 7.19.

PROBLEM SPECIFICATION

Write a menu-driven program which gives the user a list of cars that he/she can afford. The user should be given the choice of BRITISH, EUROPEAN or JAPANESE cars. The DATA statements are arranged in three groups each of which is preceded by a LABEL statement. Each data statement contains the name of a car followed by its price:

```
1000 LABEL BRITISH
1010 DATA "MINI", 4500
1020 DATA "METRO", 6200                    etc.

2000 LABEL EUROPEAN
2010 DATA "CITROEN DYANE", 4900
2020 DATA "FIAT PANDA", 5300               etc.

3000 LABEL JAPANESE
3010 DATA "DATSUN MICRA", 5400
3020 DATA "TOYOTA STARLET", 6000           etc.
```

The end of each data section is marked by:

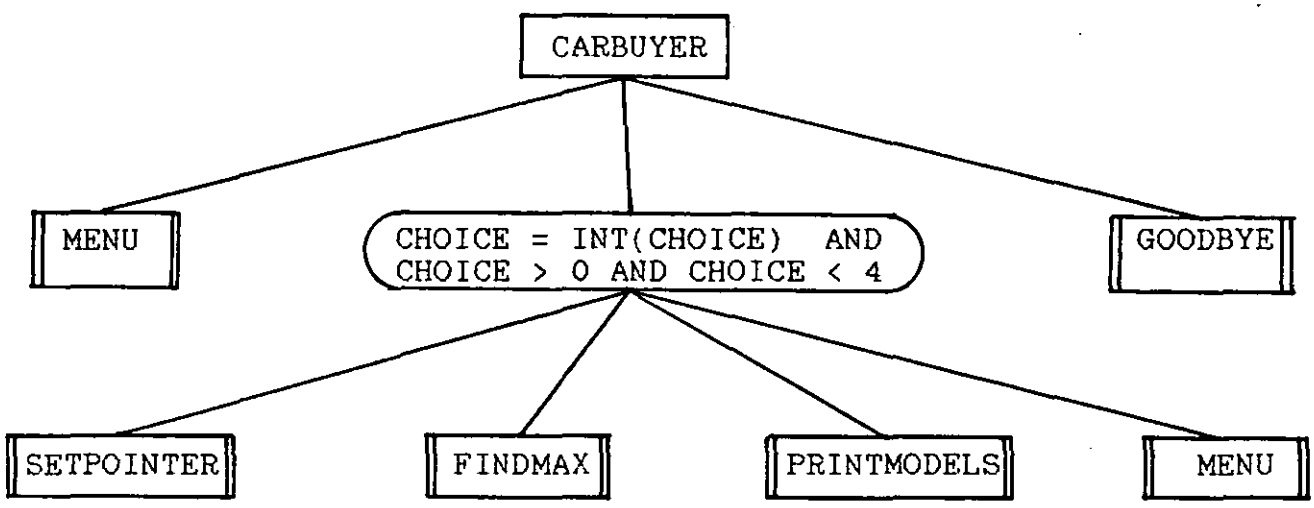
```
DATA "END", 0
```

Within each section the cars are not arranged in any particular order. The model or models that can be afforded are decided by two factors:

1. The value of the car (if any) that is being traded in. This depends on:
   a. The year it was first registered.
   b. The original value.
   c. The mileage done.
   The value is calculated as follows:
   a. 10% depreciation for each year of its age.
   b. A further depreciation of £50 for every 1000 miles in excess of an average mileage of 10000 miles per year. (if the car were 5 years old and had a mileage of 58000 this would amount to 8 * £50 = £400)
2. The user's annual income:
   The total outlay must not exceed 80% of annual income (i.e. may spend 80% of annual income in addition to any trade-in allowance).

The user should be asked to choose either BRITISH, EUROPEAN or JAPANESE from the main menu. The necessary information regarding salary and trade-in car should then be requested. The program should then output all the cars of the type requested that are within the user's price range before returning to the main menu.

FIGURE 7.18



MENU	: Displays options and accepts CHOICE.
SETPOINTER	: Sets data pointer to correct LABEL.
FINDMAX	: Calculates the maximum amount that the user can afford to spend.
PRINTMODELS	: Outputs all models of the type chosen whose cost does not exceed MAX.
GOODBYE	: Indicates termination of the program.

FIGURE 7.19

When all were satisfied that this was indeed a solution the next task was to write the various procedures. As MENU and GOODBYE were almost identical to those in earlier programs it was decided to start with SETPOINTER (figure 7.20). The use of the RESTORE LABEL statement made this very easy in COMAL. As BASIC does not support this variation of RESTORE, this section would need to be expanded if a BASIC implementation were required. This would mean that RESTORE BRITISH etc. would have to be further refined, but the overall structure would not be affected. The same would apply to data stored in files rather than in data statements.

FINDMAX was then developed (figure 7.21). This in turn called another procedure (FINDVALUE) which then had to be specified and developed (figure 7.22). This was the only section of the problem which involved a complex 'mathematical' algorithm. The final section, PRINTMODELS, was then developed (figure 7.23). This was quite a familiar type of algorithm.

The purpose of all this was to show the students how a large problem should be approached, as a preparation for their end of year project work. It was stressed that this method made it possible to analyse large problems clearly, even if a complete solution could not be achieved. For example, if a student did not know how to calculate the value of the trade-in, the rest of the problem could still be solved and then help could be sought with the difficult section. As long as the purpose of the difficult procedure has been clearly specified and it is known how the procedure fits into the overall solution, then it is perfectly valid and desirable that help should be sought in this way. If this approach is followed, then the student who has difficulties and cannot find a complete solution will at least know what questions to ask. It was hoped that if students could be taught to think like this, that they would then be able to approach any problem intelligently, rather than give up because they "didn't understand" or because they "didn't know how to calculate depreciation" etc.

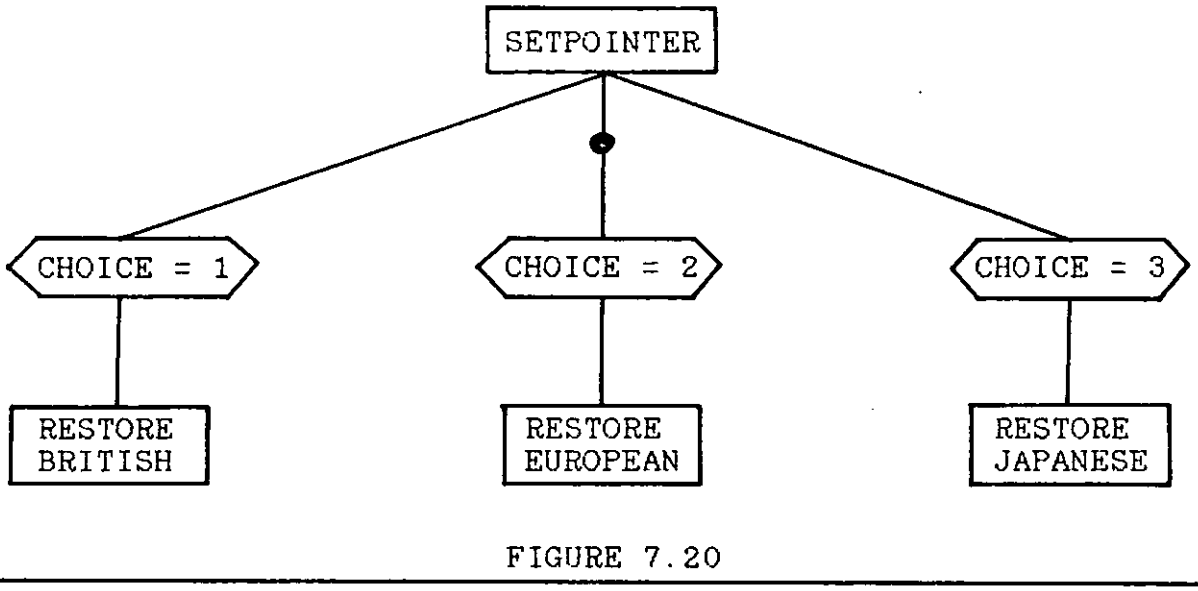
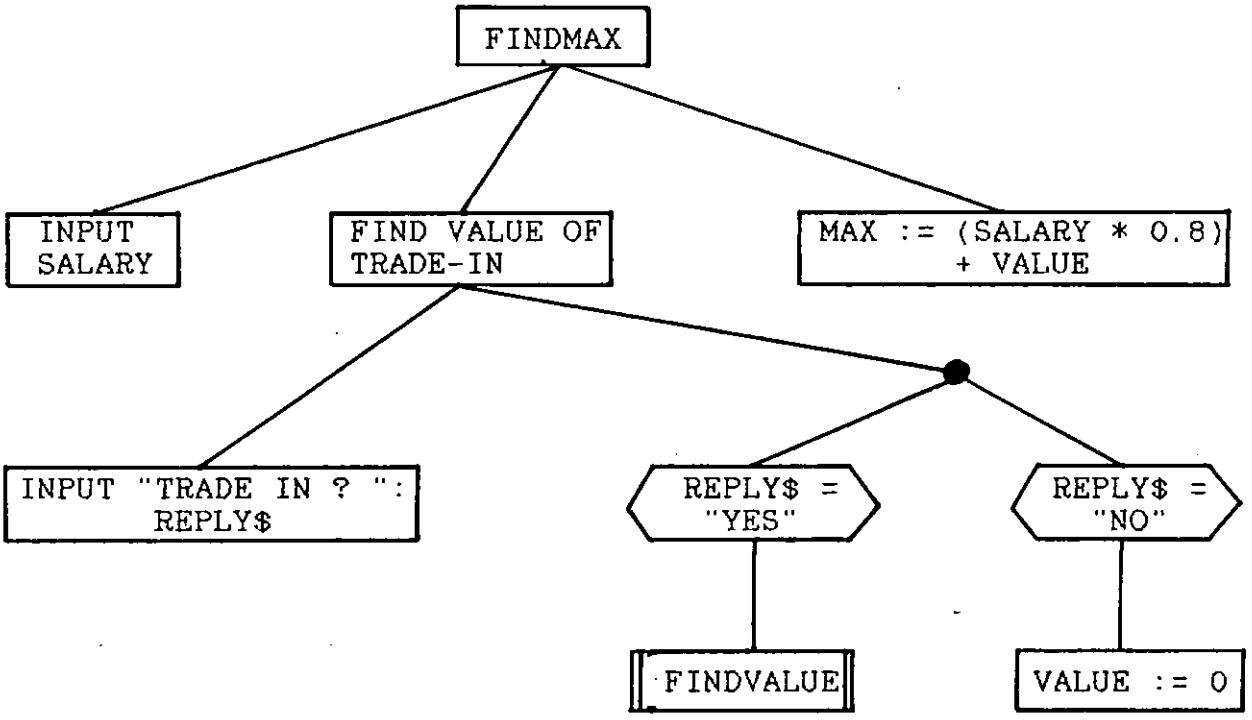


FIGURE 7.20



FINDVALUE : Calculates the current value of the car that is being traded in.

FIGURE 7.21

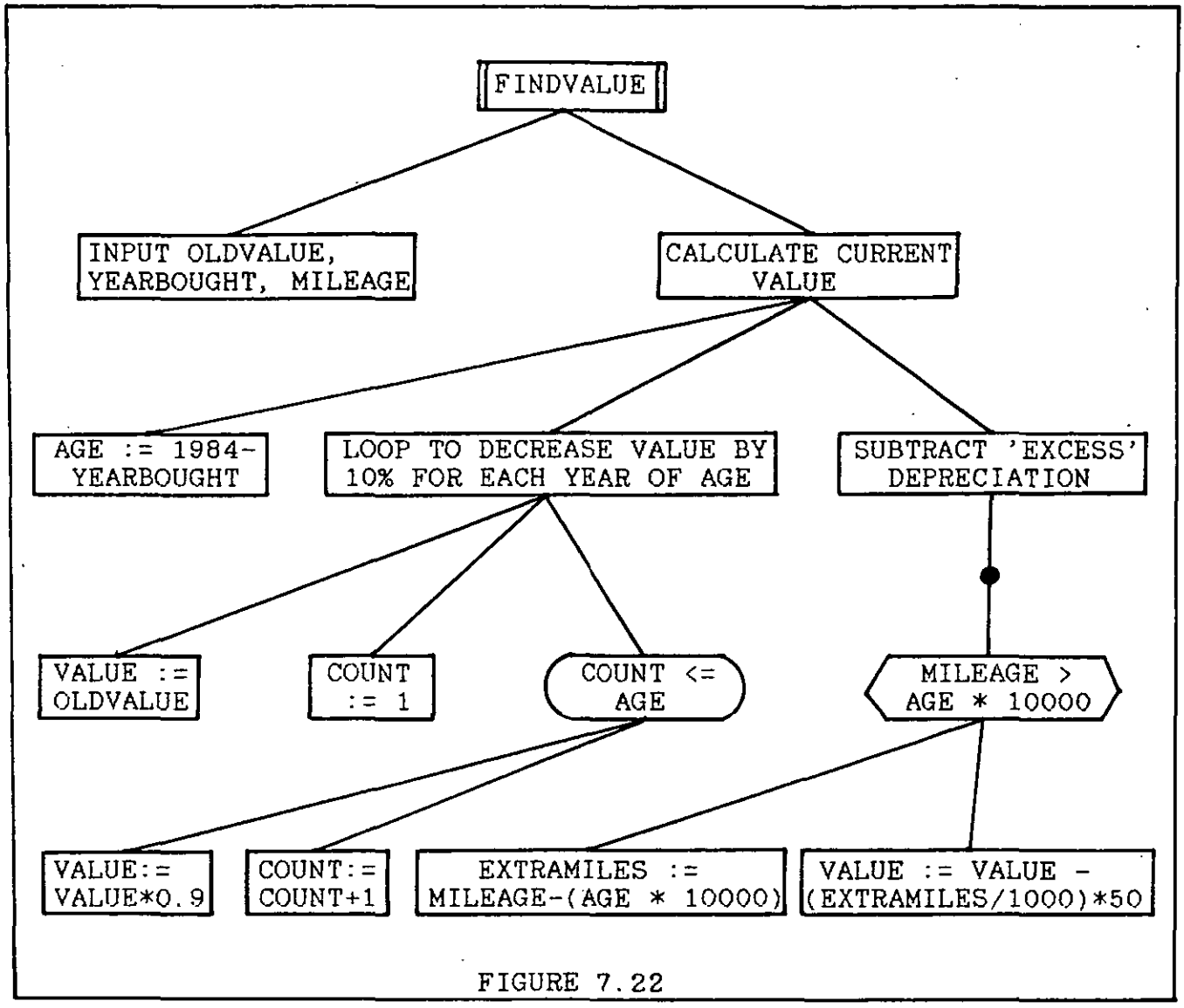


FIGURE 7.22

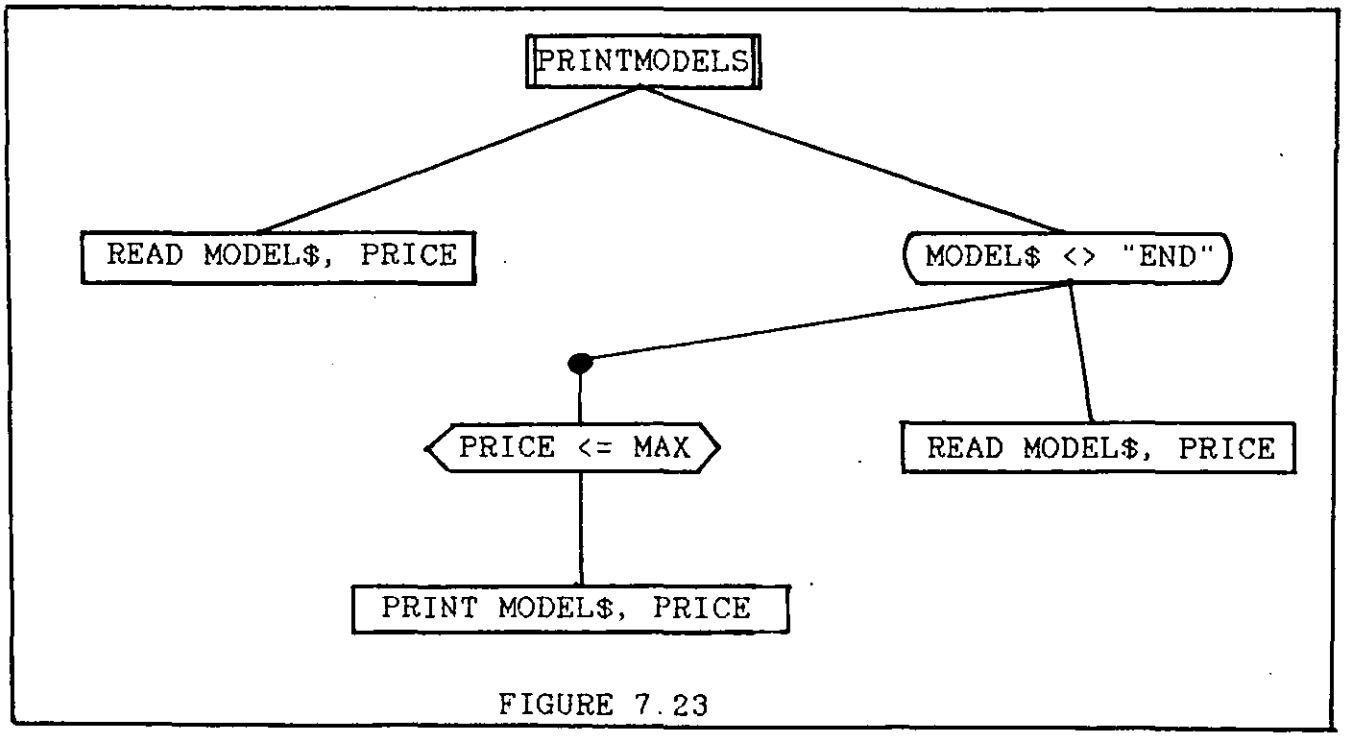


FIGURE 7.23

7.11 THE SIXTH YEAR COURSE

To begin, five problems of reasonable difficulty but not requiring any new concepts were assigned. This was to force the students to revise what they had learned in the first year. One problem was assigned per student and it was planned that each girl would present her solution to the rest of the class at a later session. All of these problems were solved very well with no more than minor bugs in any of them. Where there were bugs the students were aware of them and fully understood any corrections that were made. The class presentations were not a success, mainly due to the fact that the students were uncritical of each other's work, leaving it up to the teacher to challenge the presented solutions. This may have been due to immaturity on their part but could also have been due to their failure to recognise the importance of communicating their solution. These presentations had to be reluctantly discontinued as they were very time-consuming. This was a great pity as 'walking through' one's own solution and explaining it step by step to a critical audience could certainly help to develop confidence and would contribute to the concept of programming as a co-operative group activity.

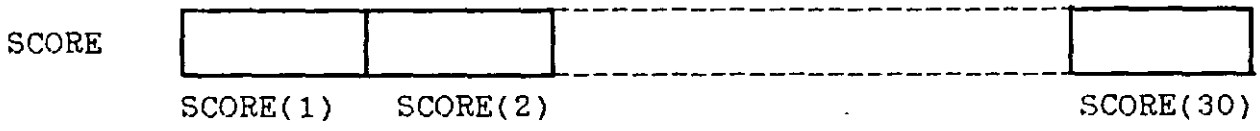
7.11.1 INTRODUCTION TO ARRAYS

A sheet containing information on arrays, along with sixteen problems, was distributed. These problems are listed in appendix A, p.216. The need for arrays was introduced using an example concerning the processing of thirty examination scores in which the number of students who failed was to be counted. This was a bad example to use as the problem could have been

solved quite simply without arrays. This difficulty is discussed in 8.11.

The convention of using a variable 'i' as the array index, when 'moving forward' through the array, was adopted. In cases where the array was being processed 'backwards' the index 'j' was used. This was a change from the normal policy of using longer variable names, but was justifiable because variable names of subscripted variables can be very long and can be difficult to read. As the convention was explained, and adhered to consistently, no problems of readability were encountered.

Arrays were depicted, like ordinary variables, as large boxes in the computer's memory but consisting of numerous different sections.



Each section was really a variable, like those that were already familiar, but the big advantage was that the array could be referred to and processed as a whole. The convention of using SCORE[1..10] etc. when referring to the array as a whole was adopted. In general A[i..j] refers to an array A whose elements are indexed from i to j inclusive. A COMAL example was given to show how a loop could be constructed to input values to an array:

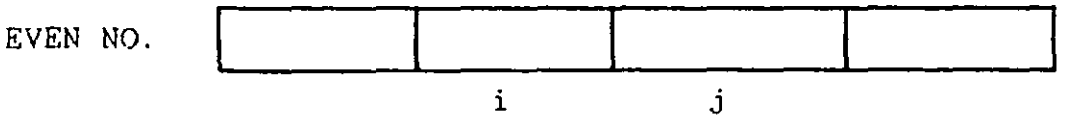
```
10 INPUT "How many values? ":NUMBER
20 DIM SCORE (NUMBER)
30 i := 1
40 WHILE i <= NUMBER DO
50    INPUT SCORE (i)
60    i := i + 1
70 ENDWHILE
```

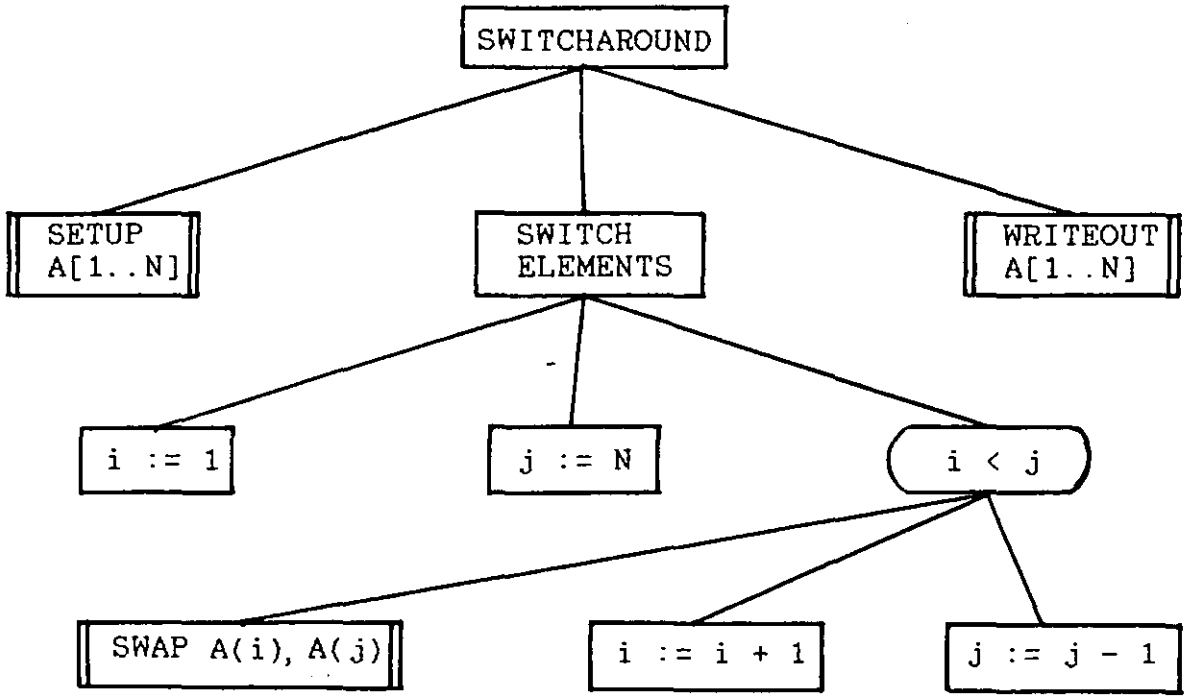
This algorithm was discussed in detail and the students were then asked to write a similar algorithm to output the values from the array, which they were easily able to do. The next few sessions were spent working through the problems on the sheet. The first ten of these were concerned with quite familiar algorithms but set in the context of arrays. These were intended to familiarise the students with the concept of an array and to give them practice in the use of the syntax associated with array manipulation in COMAL. None of these problems caused much difficulty.

The next five problems involved swapping values in an array and were slightly more difficult. For example:

> Write a program to fill an array with N numbers and then to swap the first with the last, the second with the second last etc. until the whole array has been reversed.

The solution to this problem is shown in figure 7.24. The difficult part of this algorithm is deciding on the loop guard. A number of students used the familiar i <= N guard but this causes both counters i and j to move through the whole array, causing each pair to be swapped twice and leaving the array in its initial state. The correct guard was arrived at by considering the case in which the array contained an even number of elements and the case involving an odd number of elements.





SETUP : Sets up the indicated array.

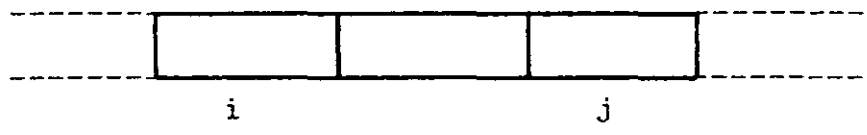
WRITEOUT : Prints out the indicated array.

SWAP : Swaps the values of any pair of variables. (A(i) and A(j) in this case).

FIGURE 7.24

In the case of an even number of elements it is clear that the processing should cease when the swaps at the indicated positions of i and j have been made. The next increment of i and decrement of j would result in i being greater than j.

ODD NO.



In the case of an odd number of elements there is no need to process the middle element and so the processing should cease when the swaps at the indicated positions have been made. The next increment of i and decrement of j would result in i being

equal to j. It follows from this that processing should cease when either i equals or exceeds j. Reversing this, the loop guard is found to be i < j.

This kind of argument is generally applicable where the array is being processed both from the front and from the back at the same time and the students were encouraged to think in terms of the two cases (odd and even) outlined above. The rest of the problems on this sheet were done very well with the exception of the one which required the first N lines of Pascal's triangle to be output. This was found to be far too difficult and no student was able to solve it.

7.11.2 STRING-HANDLING FUNCTIONS

The next group of problems involved the manipulation of strings (appendix A, p.217). This was the ideal time to introduce string handling, as strings ARE arrays and the skills acquired in the previous section could be practised in a slightly different context. Unlike their equivalents in BASIC (LEFT$, RIGHT$ etc.), the string-handling functions in COMAL clearly reflect the connection between strings and arrays:

i.e.	NAME$ (1)	is the first character in NAMES$
just as	SCORE (1)	is the first element in SCORE.

Some other COMAL string-handling functions introduced were:

1. LEN :

   This returns the number of characters in a string. It is the same as the function found in BASIC.

2. IN :

   This is an operator which tests if a given substring is present in another string.

e.g. IF A$ IN B$ THEN ....

IF "JANE" IN NAME$ THEN .... etc.

3. SUBSTRINGS :

B.B.C. COMAL allows parts of a string to be accessed in the following way:

NAME$ (i) ..... the ith. character of NAME$.

NAME$ (i:j)...all characters from ith. to jth. (incl).

NAME$ (i:).....all characters from ith. to the end.

NAME$ (:j).....all characters from start to jth.

Having introduced these functions and operators the thirteen problems on the sheet were done over a period of three weeks. Most of these problems deal with the manipulation of names. A typical example is:

Write a program which accepts a name from the keyboard in the form Christian name, space, Surname and which then outputs the same name in the form Surname, space, Christian name.

A solution to this problem is shown in figure 7.25. The crucial part of this algorithm is the loop to find the position of the space. It is important to state the name of the index of the position where the space will be found, at the second level of the diagram. This allows the rest of the solution to be found even if the section to find the space cannot be completed. This practice was encouraged throughout. Many of these problems involved searching through a string for a particular value (usually a space) and then outputting the string in a different format using the COMAL features outlined above. This search was similar to the linear search, described above, but simpler

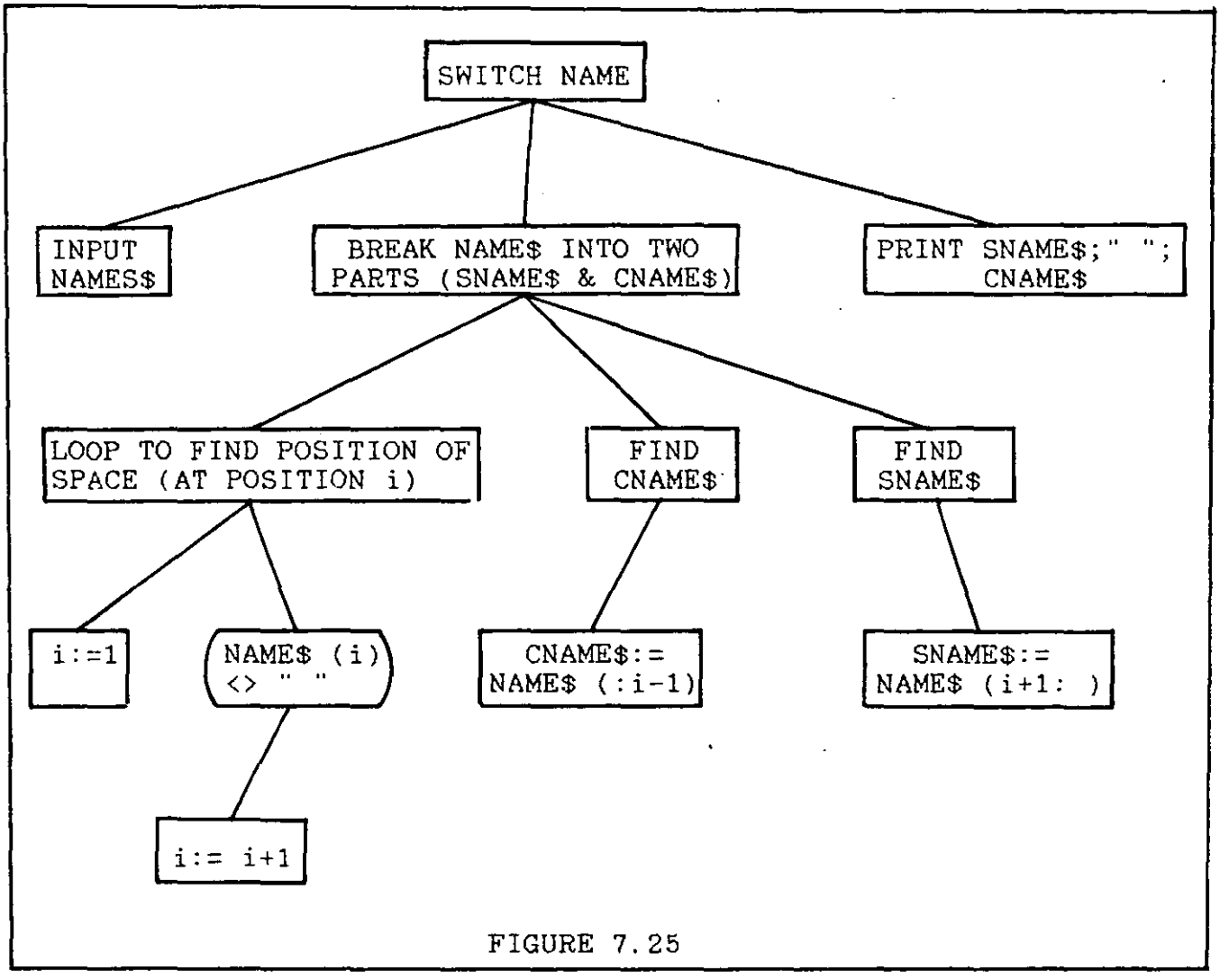


FIGURE 7.25

in that the target value (the space) was guaranteed to be present. These problems were solved very well with the exception of the last two (numbers 12 and 13) which seemed to be too difficult and caused some confusion.

7.11.3 ARRAYS OF STRINGS

The next group of problems concerned the manipulation of arrays of strings, i.e. arrays where each element is itself a string. The necessary syntax, along with eight problems, were distributed on a sheet. These problems are listed in appendix A, p.218. It was pointed out that B.B.C. COMAL distinguishes between NAME$ (1) which is the first character in the string

NAME$, and NAME$(1) which is the first string in an array of strings. The only difference is the space before the opening bracket. As with arrays, a diagram showing the concept of an array of strings as a large box with separate sections was presented.

STUDENT$(1) STUDENT$(2) etc.

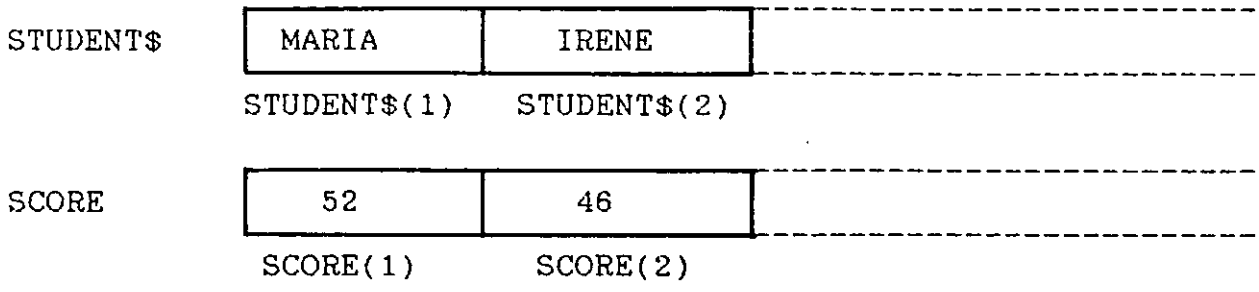
The dimension statement for this type of array needs to be explained carefully. It is necessary to specify two dimensions:

1. The number of elements (names) in the array,
   i.e. DIM STUDENT$(6) allows for six names.
2. The maximum number of characters expected in any one element,
   i.e. DIM STUDENT$(6) OF 20 allows six names of up to twenty characters each.

A typical problem of this kind is given below:

> Read ten names and ten associated scores from data into two arrays. Print the names of the students who scored above average.

This problem illustrates the idea of 'parallel' arrays which was central to some of these problems. The value in SCORE(i) is associated with the name in STUDENT$(i), so that if the score read in for MARIA was 52 and the score for IRENE was 46, the arrays could be visualised as follows:



This arrangement allows for very convenient processing and also foreshadows the concept of a record.

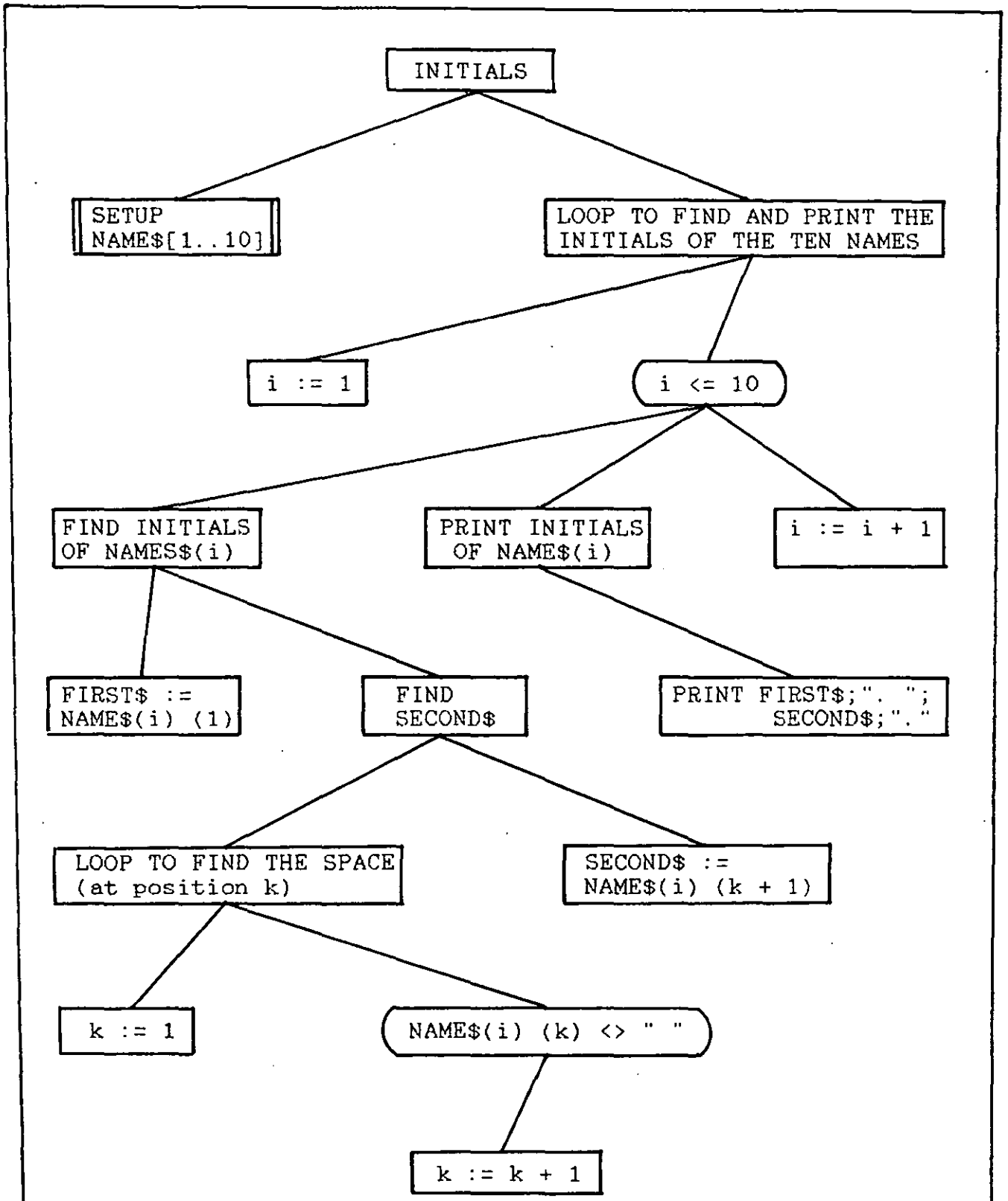
A number of the problems used at this stage involved processing lists of names. A typical example was:

> Write a program which reads ten names from data statements and then prints out the initials of each person. Each name consists of a Christian name, followed by a space, followed by a Surname.

The solution to this problem is in figure 7.26. This set of problems was quite difficult but they all had the same general outer structure:

```
100 i := 1
110 WHILE i <= N DO
120   PROCESS NAMES$(i)
130   i := i + 1
140 ENDWHILE
```

This outer loop controls the 'movement' down through the array taking each name in turn. Once this has been established, the student is then free to concentrate on refining the 'PROCESS NAME$(i)' statement. In each case, the algorithm for processing NAME$(i) was already familiar from the previous section on string variables. This meant that most of the problems could be solved by superimposing this outer structure on to the previous solutions. There were also some syntax considerations in implementing solutions to these problems. The first bracket




```
DATA "MARIA CORRIGAN"
DATA "IRENE COMERFORD"    etc. (10 such DATA statements)
```

FIGURE 7.26

after the name of the array contains the index of the required element, while the second bracket contains the index of the required character within this element. Thus NAME$(i) (k) refers to the kth. character in the ith. element of the array. The second bracket may also contain two parameters, so that NAME$(i) (j:k) refers to the group of characters, starting at the jth. and ending with the kth., of the ith. element of the array. This caused quite a lot of difficulty and perhaps some exercises on this syntax should have been done before attempting to use it in difficult problems.

7.11.4 SORTING ARRAYS

The bubblesort algorithm was used to introduce the topic of sorting, as it was felt that this was the simplest and most accessible of the sorting algorithms. It had been intended to investigate some other sorting processes but this was impossible due to time constraints. The outline of the bubblesort algorithm was described and then the first pass on a list of four numbers was demonstrated using the following algorithm: "Starting at the left, compare each value with its immediate neighbour (to its right) and if it is found to exceed its neighbour then swap the two values".

6	3	5	2	(6 > 3 therefore swap values)
3	6	5	2	(6 > 5 " " ")
3	5	6	2	(6 > 2 " " ")
3	5	2	6	(end of first pass)

After working on a similar list of numbers the students were shown a structure diagram for this process (figure 7.27).

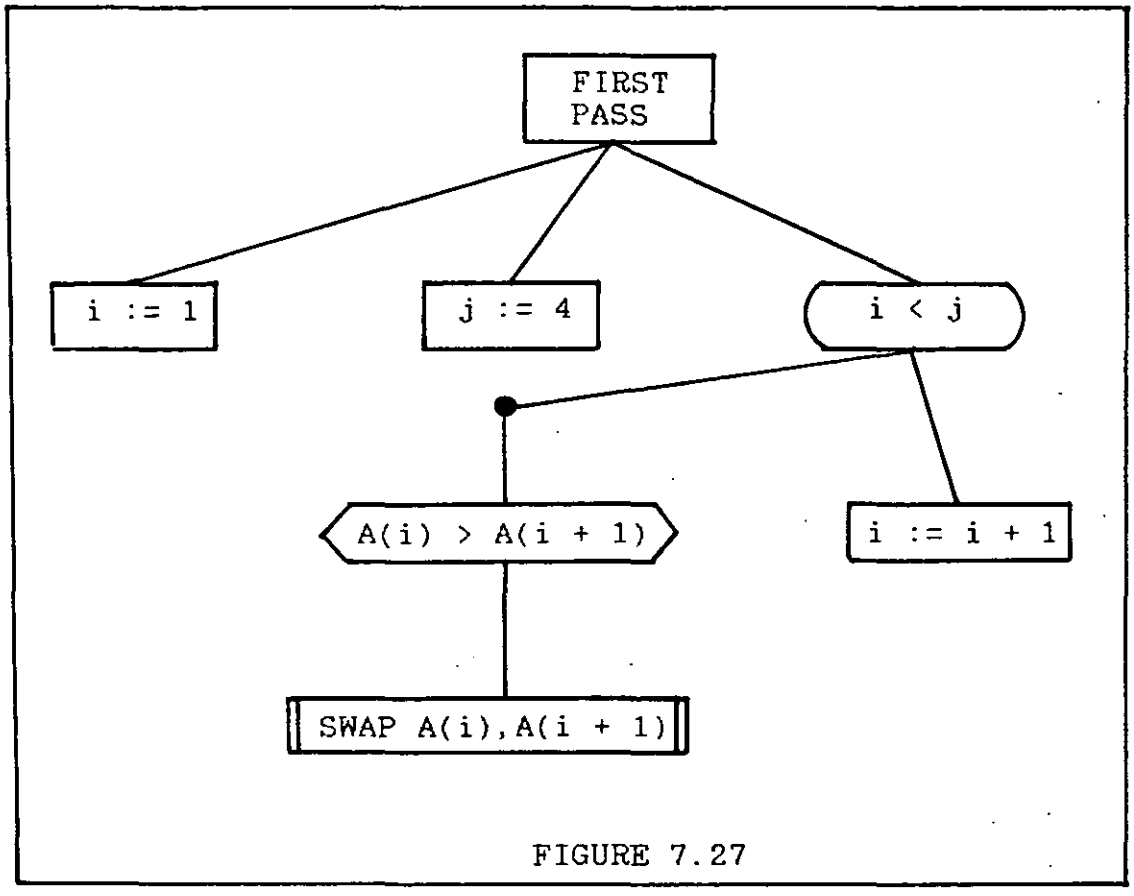


FIGURE 7.27

The loop guard i < j is used here instead of the more familiar i <= j because the pass is complete when i has reached the second last number in the list. If i were allowed to become equal to j, then there would be no number to the right for comparison, i.e. there is no A(j+1). It was agreed that the last element in the list could now be forgotten about as it was already in its correct position. The next task was to process the first three elements in the same way as above and then, finally, the first two elements. The students were required to do this manually. When this had been done they were given several different lists to process manually before being shown the complete bubblesort algorithm (figure 7.28).

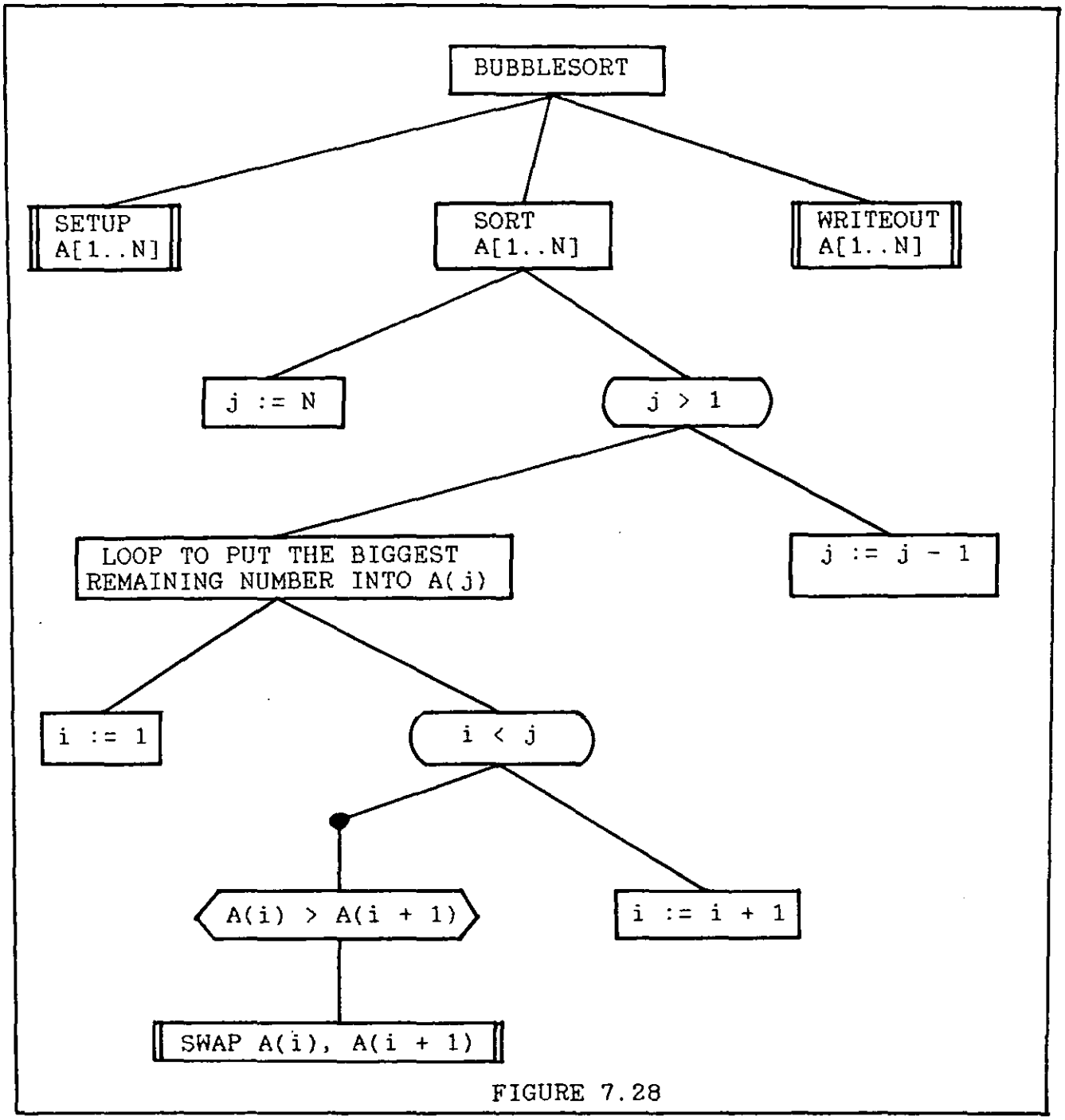


FIGURE 7.28

An exercise requiring the use of the bubblesort algorithm was then given. This required the use of the supplied algorithm, to give the user the choice of outputting a list of names, in either alphabetical order or order of merit (based on supplied scores) and was done very well. Another exercise done at this stage was to use the internal clock of the B.B.C. microcomputer to investigate the efficiency of the bubblesort algorithm. This

involved using the clock to time bubblesorts for arrays of different sizes. The inefficiency of this algorithm is not at all obvious when processing arrays of twenty or thirty numbers, which require just a few seconds. Its inefficiency was demonstrated very clearly, however, when it took over an hour to sort an array of one thousand elements.

CHAPTER 8

EVALUATION OF THE COURSE

8.1 INTRODUCTION

In this chapter some difficulties that arose concerning certain sections of the course are outlined and the adjustments made for the second implementation of the fifth year course are described. This is followed by a review of the projects undertaken by the students. These are on the accompanying disk and are a good measure of the effectiveness of the course. The chapter terminates with the description of an application of structure diagrams in Leaving Certificate mathematics.

8.2 THE METANIC COMAL SYSTEM

Apple Metanic COMAL was used as the implementation language throughout the fifth year course and was found to be generally satisfactory with just one serious flaw. To access this version it is first necessary to get the machine into CP/M and then to call up COMAL. CP/M and COMAL are loaded from the same disk. In the first year each student was given a CP/M formatted blank disk for storing COMAL programs. Each student used the same master disk to start up the system, inserting her own disk after the system was in COMAL. This caused some problems due to a quirk of Metanic COMAL which requires INIT to be typed every time a disk is changed. If this is not done, and students

often forgot to do it, it is impossible to save a program on the new disk. This is a serious flaw in the system and there is no way to recover once the error has been made. This caused a number of programs to be lost which was very frustrating for the students concerned. Because of this, in the second year each student was given a disk containing both CP/M and COMAL. This meant that there was no need to change disks after COMAL had been brought up. There was, of course, less space available on the students' disks for their own programs but this did not cause any problems.

8.3 THE CONCEPT OF A COMPUTER SYSTEM

All students in both years developed the ability to use the computers confidently and unsupervised. They were all able to load and save programs, edit programs and ENTER library procedures into programs without difficulty. This indicates that the simple mental model used was satisfactory and that there was no need to describe the system configuration in more technical terms.

8.4. VARIABLES

All students acquired the concept of a real variable reasonably quickly. However there was some confusion in relation to the other types of variable. COMAL supports four distinct types of variable: real, integer, string and boolean. In the first year of the course all four were described but it was found that the distinction between real and integer variables caused difficulty

for many students. It was also found that, using the mini-language, there was no need for boolean variables. For these reasons it was decided to introduce only real and string variables for the second year. Real variables were always referred to as numeric variables. As all the algorithms used in the course may be written using just real and string variables, there is no good reason for introducing the others if they are likely to cause confusion. Omitting them proved to be very satisfactory.

8.4.1 STRING VARIABLES

In the first year, very little explicit instruction was given on string variables. They were initially introduced at the same time as real and integer variables. Numerous examples concerning the use of real variables were used but very few examples involving string variables were demonstrated. This was because it was considered that the concept of a string variable was essentially the same as that of a real variable, the only difference being in the syntax of assignment and the need for dimensioning. A sheet containing all the necessary information on string variables was distributed to each student but the topic was not dealt with in detail in class. The students were requested to study the sheet themselves and then to keep it for reference. This was a mistake. The students tended to dismiss material treated in this way as being of lesser importance than that spelt out in class and paid little heed to it.

To overcome this problem, a whole session was devoted to string variables in the second year. More comprehensive sheets were prepared (appendix A, p. 208). These sheets contained the rules for string variable names, followed by the syntax for both the assignment and input statements. When some examples on these statements had been carried out by the class, the need for dimensioning was raised. It was simply pointed out that string and numeric values were stored in different ways by the computer. No matter how large a number is, it only requires a fixed amount of space because it may be rounded off. Strings, on the other hand, should not be rounded off, so large strings require large amounts of memory. Therefore in the interests of economy, it was necessary to specify how large each string variable might be, so that the computer could set aside a suitable amount of memory space. This explanation of the need for dimensioning was accepted with some misgivings by the class. Another change in the treatment of string variables was to require students to run programs which utilised them, before the topic was discussed in class. This prepared them for the idea of a non-numeric variable, as they could see that the computer could 'remember' words, names etc. Despite this extra effort, students still had difficulty with string variables. The two most common mistakes were to omit the DIM statement and to use a numeric variable where a string variable should have been used. All these problems were, however, overcome with practice.

8.5 THE INPUT STATEMENT

It was noticed during the first year that some students had severe difficulties with the input statement. There was some confusion between what appears on the screen when the program is listed and what appears when it is run. The main error was to attempt to give a value to the variable when keying in the program.

For example : INPUT LENGTH 6

or INPUT 6

For this reason, during the second year, students were required to run numerous teacher-written programs, which utilised input statements, before the topic was mentioned in class. The students enjoyed using these programs, especially the ones which asked for their names and then carried on a 'personal' conversation using the supplied name. They became curious about how the programs worked and how numbers and names were 'remembered' by the computer. They were therefore better prepared to acquire the concept of input when it was introduced. The difficulties mentioned above did appear again in the second year, but much less frequently, and were cleared up much more quickly.

It was found that the students who spent most time at the machines in these early stages were least likely to encounter difficulties of this nature. This seems to suggest that practice at the machines is very important in the early stages

and may be more beneficial than extensive class instruction, supplied notes etc., at this stage.

8.6 TYPES OF PROBLEMS FOR EXERCISES

In general, the problems that were assigned in the early stages were very simple and concerned topics that were familiar from other subject areas. However, in the first year, some examples were assigned involving topics which were not already familiar to the students (e.g. A.P.s, G.P.s, Compound Interest etc.) but which were really quite simple and which were included in their fifth year mathematics course. This was a mistake and confused a number of students. In fact, some of them were still confused even after solutions had been shown and explained in detail. It would seem, therefore, that in the early stages of the course, it is unwise to ask students to solve problems from areas which are not extremely familiar. This might also be an argument against running programming and mathematics courses 'in parallel'. It seems that they had enough to cope with in learning COMAL syntax and how to operate the machines without having to solve problems from 'new' areas. These problems were not assigned in the second year.

8.7 STRUCTURE DIAGRAMS

The students initially resisted the idea of using diagrams and were very slow to accept that, when drawing a diagram, certain 'syntax' rules needed to be observed. For example, it was very common to find a sequence of statements listed vertically rather

than horizontally; i.e. if a certain task, A, required the execution of three steps, 1, 2 and 3, it was often drawn as in figure 8.1, rather than as in figure 8.2. This kind of error ruined the meaning of the diagram and spoiled the Top-Down approach.

A lot of difficulty was initially encountered in trying to convince the students that it really mattered how the diagrams were drawn. They seemed to feel that it was sufficient to derive a program that worked (or seemed to work), either with a badly drawn diagram or with no diagram at all.

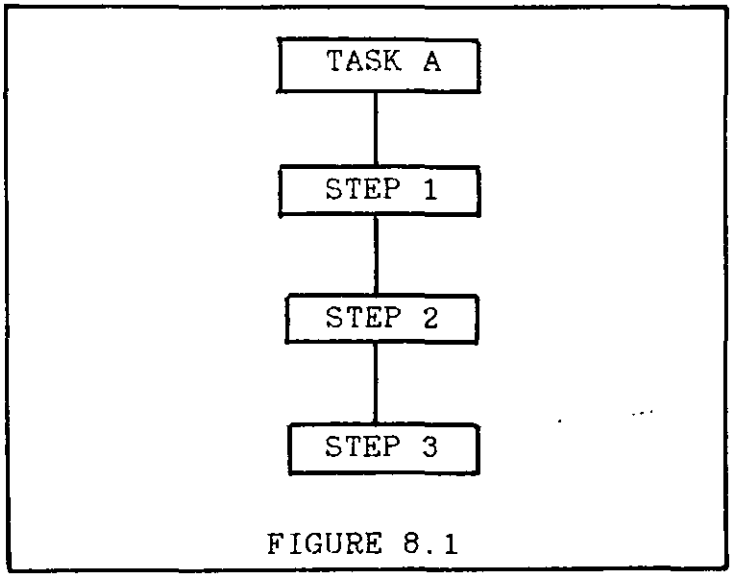


FIGURE 8.1

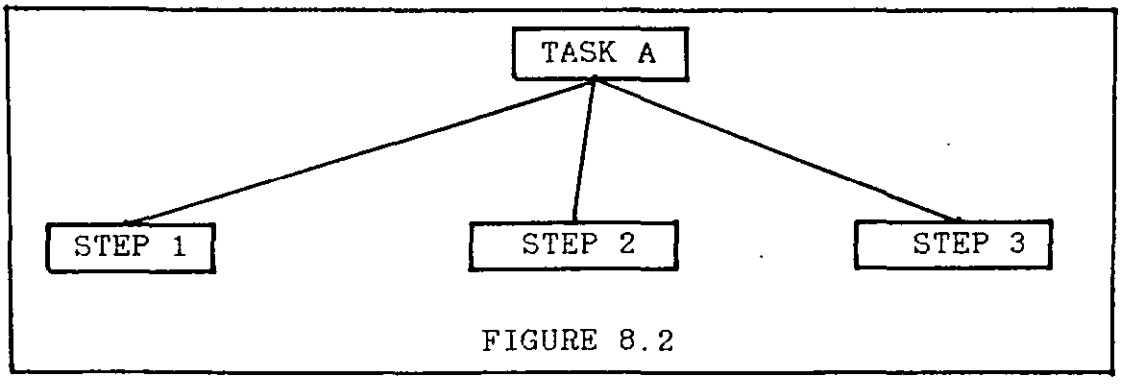


FIGURE 8.2

It was also noticed in practically all cases, whether the diagram was organised properly or not, that they were very sloppily drawn and presented. Very few students seemed to use a ruler. This was despite the fact that all the diagrams they had been shown, whether on overhead slides or on typed notes, were very carefully drawn. It seems they were unable to appreciate that a well drawn diagram could be an effective means of communicating a solution. In fact, it is possible that they didn't appreciate that there might be a need to communicate a solution at all, other than to a machine. It is understandable that the students had difficulty in accepting the idea of using structure diagrams. The kind of problems encountered at this stage are usually simple and straightforward enough to be coded directly by most students. To the novice, the idea of drawing a diagram may seem to be a waste of time. If, however, the introduction of diagrams is postponed until really big problems are encountered, students who solve problems at the keyboard will find it very difficult to adapt. There is also the likelihood that the combination of quite difficult problems and a new method of approach might overwhelm the students and discourage them.

Initially, many students complained that they found the diagrams 'very difficult'. It was subsequently discovered that they could all interpret the diagrams quite well, but had difficulty in designing them. This seems to indicate that it was the

problems rather than the diagrams that they found difficult, but were unable to distinguish between the two. Despite these difficulties, the students came to appreciate the diagrams and in later sessions, when they were given the choice of using the diagrams or of coding directly, all chose to use the diagrams.

8.8 THE CONDITIONAL CONTROL STRUCTURE

One unexpected difficulty that arose in the early stages concerned the output statements from simple programs involving a conditional structure. This may be illustrated by considering the following problem:

> Input a number at the keyboard and say whether it is even or odd.

There are two ways in which the output from this type of problem can be organised and these are illustrated in figure 8.3 and figure 8.4. The solution in figure 8.3, although it is slightly longer, is better, as the first level of the diagram breaks the problem down into three separate sub-tasks representing INPUT, PROCESS and OUTPUT. This is a sub-division that occurs in numerous problems, including many quite complex ones. On the other hand, there are situations in which this method cannot be applied. An example of this is the problem of outputting the greater of two numbers (see figure 7.10). If the numbers happen to be equal, then the output should be a message saying that they are the same (a string). The output may therefore be either numeric (one of the numbers) or string. In these cases

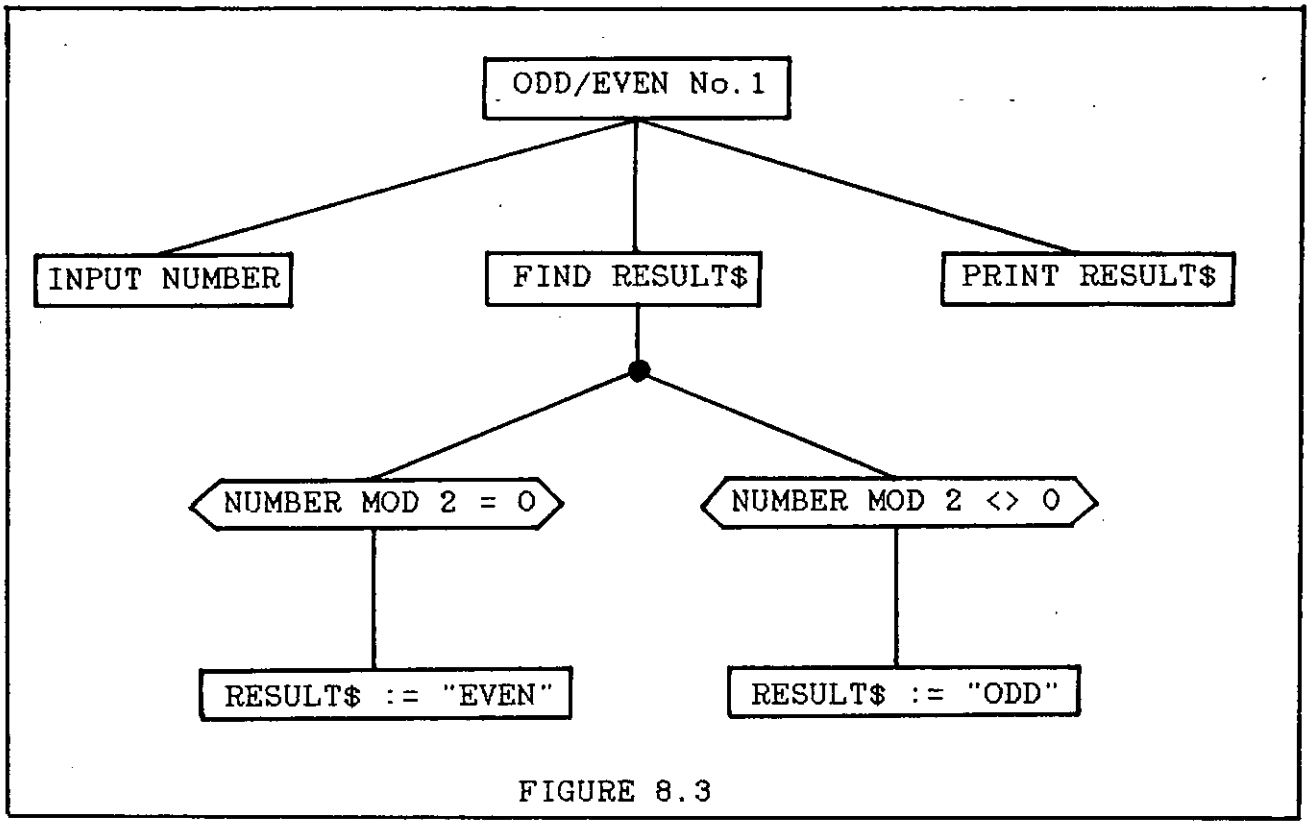


FIGURE 8.3

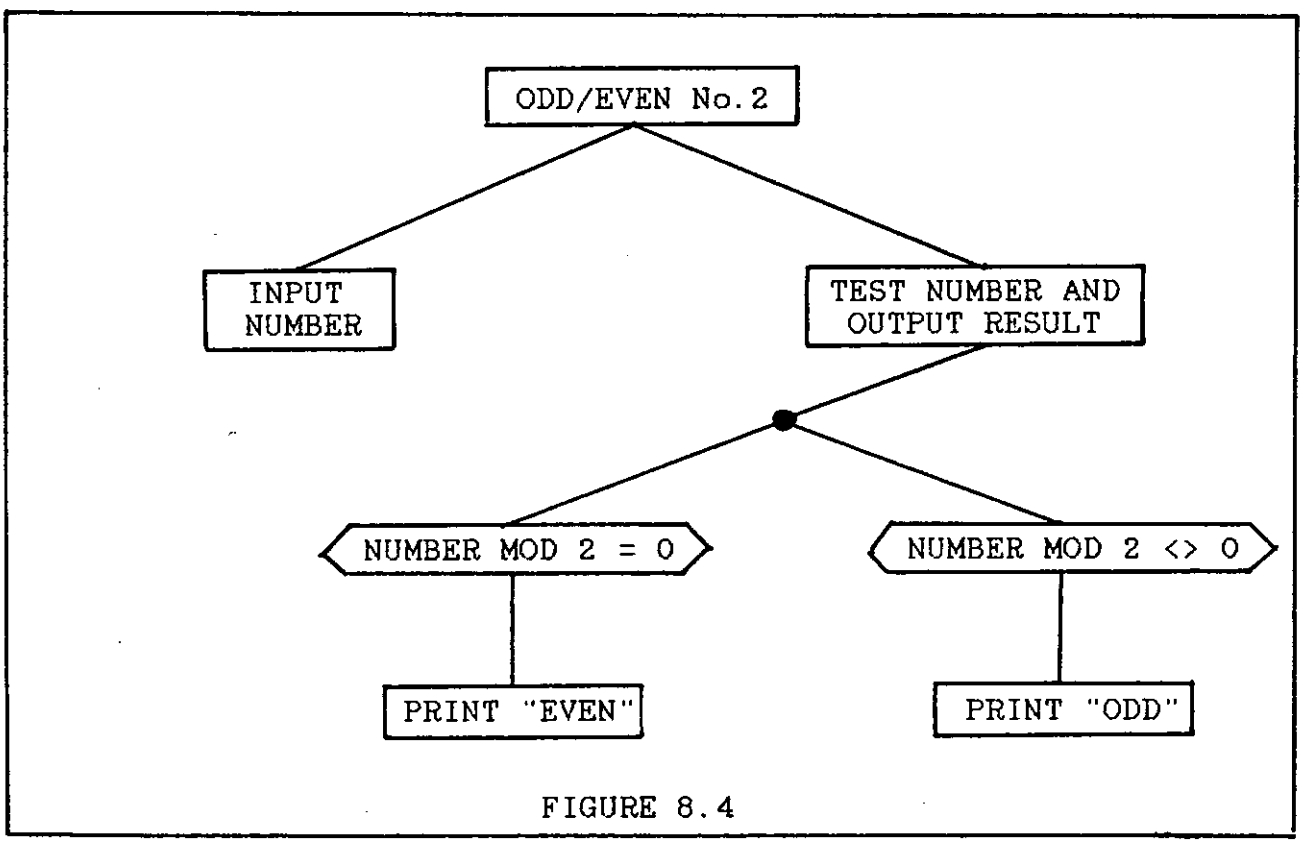


FIGURE 8.4

the alternative method (figure 8.4) should be used. Students were encouraged to use the first method wherever possible but were also shown some examples in which the second method was used. This led to some confusion and a common error encountered in the early stages was the inclusion of two sets of output statements. As more practice was acquired the confusion was gradually dispelled. However, in the second year, numerous problems of the first kind were used, before any of the second kind were introduced, and this was quite satisfactory.

8.8.1 THE USE OF AN ELSE BRANCH IN IF STATEMENTS

In the first year the ELSE branch was introduced as an extension of the conditional control structure. An information sheet was distributed and examples were given in which ELSE might be used. It was stressed that the use of ELSE was never strictly necessary and that it should be used with extreme caution. It was suggested that it should only be used in cases where the final condition of the IF structure was obvious, but long and tedious to compose. The following program fragment, which decides whether a given letter (A$) is a vowel or a consonant, was given as an example:

```
100 IF A$="A" OR A$="E" OR A$="I" OR A$="O" OR A$="U" THEN
110    PRINT "IT IS A VOWEL"
120 ELSE
130    PRINT "IT IS A CONSONANT"
140 ENDIF
```

The alternatives to ELSE in this program are quite long expressions and so it was suggested that ELSE might be used in

circumstances such as this. The class was shown how to construct a diagram for a conditional structure including an ELSE branch, and how to convert this into COMAL.

It was noted that the students rarely used an ELSE branch in their own work and often made errors when they did use one. It was realised that the use of ELSE in anything other than two-branch conditional statements usually resulted in a loss of clarity. The reason for this is that to comprehend the circumstances under which the ELSE branch is executed, it is necessary to 'back up' through all the previous conditions. It is also necessary to reverse each one, and mentally combine the reversed conditions. This is quite difficult but must be done to understand such a program. If this is not done it is very easy to overlook special cases, causing the ELSE branch to be executed in error. For this reason ELSE branches were not introduced at all in the second year of the course and this was much more satisfactory.

8.8.2 THE USE OF 'EXPLICIT' BOOLEAN CONDITIONS

In the first year it was noticed that the students composed the boolean conditions of conditional statements in an 'explicit' manner most of the time and that their programs were clearer and less error-prone when they did this. An example will illustrate the point:

Write a program in which the number of children per family is input and which then decides how many pints of milk are required under the following rule:

No children	.....	2 pints
1, 2 or 3 children	.....	4 pints
4 or 5 children	.....	6 pints
6 or more children	.....	3 pints + 1 per child

The normal textbook approach to coding the conditional statement in this problem in COMAL is as follows:

```
100 IF KIDS = 0 THEN
110    PINTS := 2
120 ELIF KIDS < 4 THEN
130    PINTS := 4
140 ELIF KIDS < 6 THEN
150    PINTS := 6
160 ELSE
170    PINTS := 3 + KIDS
180 ENDIF
```

In order to understand the circumstances under which the third branch of this structure is executed, it is not sufficient to examine the boolean condition guarding it (KIDS < 6). This condition is TRUE when KIDS has any value less than six but, of course, the branch is only executed when the value of KIDS is four or five. To realise, this it is necessary to 'back up' through the structure examining each of the previous conditions. This makes the program difficult to read and to understand. The final ELSE branch is an extreme case of this lack of clarity.

Using 'explicit' boolean conditions the following program fragment is derived:

```
100 IF KIDS = 0 THEN
110    PINTS := 2
120 ELIF KIDS = 1 OR KIDS = 2 OR KIDS =3 THEN
130    PINTS := 4
140 ELIF KIDS = 4 OR KIDS = 5 THEN
150    PINTS := 6
160 ELIF KIDS >= 6 THEN
170    PINTS := 3 + KIDS
180 ENDIF
```

In this case the total condition governing each branch is explicitly stated at the entry point to the branch and this results in a clearer, more readable program. It may also be noted that this solution is a much closer reflection of the problem statement and therefore is easier to check and debug. It was decided to insist on explicit statements such as these for the second year of the course and this was successful.

In order to write these explicit conditions, it is usually necessary to use boolean operators. However, it was felt that boolean operators should not be introduced until the concept and syntax of the conditional statement were reasonably well established. This meant that it had to be possible to solve all the initial IF-type problems with explicit conditions, but without boolean operators. It was difficult to find problems of this nature which contained more than two branches. Five of the eleven problems which were used at this stage in the first year had to be postponed until after boolean operators were dealt with. Eleven problems were eventually used and these are in appendix A, p.215.

8.8.3 De MORGAN'S LAW

Most of the class avoided using the NOT operator by turning expressions around, i.e. using A >= 3 rather than NOT A < 3. Those who did use NOT tended to get confused when constructing expressions with more than one condition. This difficulty with the NOT operator prompted the introduction of de Morgan's law in the second year. In all cases where it was necessary to reverse a condition, de Morgan's law was used rather than the NOT operator, although the NOT operator was still introduced for the sake of completeness. The principal reason for this was that many, if not all, expressions which involve the NOT operator are difficult to understand. At the simplest level it may be argued that X > 10 is much easier to comprehend than NOT(X <= 10) which means exactly the same thing. When used with large expressions, NOT can almost render them incomprehensible. For example:

```
NOT ( X <= 6   OR   X MOD 5 <> 0 )
```

is equivalent to

```
X > 6   AND   X MOD 5  = 0
```

In the latter case it is obvious that the expression is TRUE when X is a multiple of 5 but greater than 6, i.e. it is TRUE when X has one of the values 10, 15, 20, 25.....etc.
This is not at all clear from the former expression. It would appear that the NOT operator is not only confusing but redundant, as any boolean expression can be written without it by using a combination of ANDs and ORs.

De Morgan's law is used to reverse any expression as follows:

1. Reverse all the component conditions.

   = becomes <> and vice versa.

   > becomes <= and vice versa.

   < becomes >= and vice versa.

2. Change all ORs to ANDs and vice versa

For example the reverse of the condition

X <> 5 AND (Y = X OR Y <= 2)

is X = 5 OR (Y <> X AND Y > 2)

In the first year, the most frequent use of NOT was to reverse boolean expressions to cater for either/or situations. These situations occur quite frequently as in the program to simulate throwing two dice (see 7.7.2). If the program is to be altered to output a message whether a prize has been won or not, then a two-way branch is required. The guard for the second branch may be derived in any of three different ways:

1. Use the NOT operator.
2. Use ELSE.
3. Use de Morgan's law.

All three of these methods are quite simple to apply even if de Morgan's law requires slightly more effort on the part of the programmer. However, it is felt that the resulting program is always much clearer and easier to understand, if this method is applied. Students had no difficulty in learning the method and completed numerous examples accurately in class. The only difficulty found here was in convincing the students that it was

worth the slight extra effort. There is a case here for omitting the NOT operator altogether and this is being seriously considered for future courses.

All the students became proficient in the use of the conditional statement and the construction of boolean conditions. The fact that they mostly avoided the use of ELSE, and that they generally made their conditions explicit, was very encouraging and showed that they appreciated the need for clarity. The omission of the CASE statement did not cause difficulty as there was no need for it in any of the problems on the course.

8.9 THE ITERATIVE CONTROL STRUCTURE

No serious difficulties were encountered with this part of the course. The students were already very familiar with boolean conditions when this section was introduced. They therefore had little difficulty constructing loop control conditions, which is the most important skill in loop construction. The guidelines that they were given for constructing loop conditions were found to be helpful, as was the distinction that was drawn between fixed and indefinite iteration. The fact that only one looping structure was used caused no problems and there was never any need for structures other than the WHILE loop.

8.10 PROCEDURES

When this work was originally planned it was intended to introduce procedures at a very early stage because it was felt

that procedures were a fundamental part of structured programs. However, it was realised as the work progressed that it would be pointless to introduce procedures until the students were able to tackle reasonably large problems in which the use of procedures was justified. Procedures could have been introduced earlier, but the students would not have learnt any more than the syntax of the EXEC, PROC and ENDPROC statements, had this been done. Although procedures had not been introduced, the use of a strict Top-Down method had encouraged the students to think 'procedurally' from the start. The use of the diagrams forced students to think about problems in high level terms before details were considered. This is the essence of 'procedural' thinking and was well established early in the course. The actual use of procedures in problems and the associated syntax was easily assimilated by all the students and they all came to use them well, as is evidenced by their projects (see 8.12).

8.11 ARRAYS

The main difficulty associated with the introduction of arrays was to find suitable introductory problems in which their use was justified but which were not too difficult. The initial problem used was quite unsatisfactory (see 7.11.1) as it did not really require the use of an array and this was realised by some of the students. The problem was to decide how many students from a class of thirty had failed a test (i.e. scored less than forty marks). The teacher then suggested that without arrays

this would require thirty separate variables, SCORE 1, SCORE 2 etc. and thirty different conditional statements:

```
IF SCORE 1 < 40 THEN        etc.
```

This was not correct, as the problem could have been solved very simply without arrays as follows:

```
 10 FAILED := 0
 20 COUNT := 1
 30 WHILE COUNT <= 30 DO
 40   READ SCORE
 50   IF SCORE < 40 THEN
 60      FAILED := FAILED + 1
 70   ENDIF
 80 COUNT := COUNT + 1
 90 ENDWHILE
100 PRINT FAILED
```

What is required at this stage is an example in which an array is really necessary and this usually implies that the data must be processed twice. This would involve changing the above problem as follows:

> Write a program to accept thirty scores from the keyboard and find the number of scores that are above the average.

In this case each score is first used to calculate the mean and then, when this is done, each score must be compared with the mean. Therefore each score must be stored after it is initially used to calculate the mean. In this case the choice is between thirty separate variables (and thirty IF statements) or a thirty element array. This point is considered to be important because it is known from experience that many students have difficulty in deciding when it is appropriate to use an array, and tend to

use arrays in situations where they are not appropriate. It is important that the first few problems encountered in this section should demonstrate the real need for arrays and not just the syntax associated with them. Some other problems which are quite simple, which require arrays and which will be used in future courses are:

1. Input a set of 10 numbers and then print them in reverse order.
2. Input a set of 10 numbers and then print those that are bigger than the last number that is input.
3. Read a set of numbers from data and find their standard deviation.
4. Read a set of numbers from data. Ouput either the even numbers or the odd numbers of the set, whichever has the greater sum.

Apart from this initial difficulty, the use of arrays did not cause any great problems for the students, although some had difficulty with the syntax in the early stages. The sorting and searching algorithms used in the final stages were also found to be difficult by some students. This is understandable as they are quite difficult and the students were under a lot of exam pressure from other subjects, leaving them little time to devote to these problems.

8.12 PROJECTS

Towards the end of each of the fifth year courses the students were required to undertake projects. A decision had to be made whether to give the students a free choice of project or to require them to undertake specific, well defined-problems. The advantage of the former was that each student could choose a topic which was of interest to herself. On the other hand, if this were done there would be difficulties of problem definition. Some students would probably want to attempt too much, while others would be inclined to do as little as possible, and still more would probably not be able to think of a project at all. Because of this, it was decided to require them to work on specific, teacher-defined problems.

A list of big and quite difficult problems was drawn up each year (appendix C, p.228). The students were given the option of working alone or in pairs. Most students chose to work in pairs and were allowed to select their own partners. When the groupings had been decided, the problems were allocated according to the abilities of those in the group. Thus the two most able students in the class were assigned the most difficult problem and the weakest student, who chose to work alone, was assigned the easiest one. It was hoped that each problem would be difficult enough to constitute a challenge for those involved, without being so difficult as to discourage them. They were advised to approach the problems in the way that was

outlined above in the CARBUYER example (see 7.10). To encourage this approach, they were required to submit the main program and the procedure specifications for scrutiny, before going on to develop the lower levels of the problem. They were generally reluctant to do this but, after some persuasion, they eventually did it quite well. It was intended to carry this approach through for each level of each problem, with all procedures being submitted as they were developed. This proved to be impractical due to school holidays, absenteeism, excuses etc., and was not attempted in the second year. Despite this, the final solutions achieved were very good in both years. The supplied disk contains some sample projects. All of these programs have names that end with PROJ (e.g. SUMSPROJ).

A good example of the projects submitted was the simulation of the game of NIM (NIMPROJ.CSB). A full listing of this program is in appendix C, p.230. The problem definition was as follows:

> The game of NIM is normally played by two players. Starting with any number of matchsticks, each player is allowed to remove 1, 2 or 3 at a time. The player left with the last match loses. The 'trick' is to leave your opponent with 5 matches in which case, no matter how many she takes, you can always ensure that she is left with the last one. To make sure that she is left with 5 you should ensure that she is left with 9 (13, 17, 21, 25 etc.). Write a program in which the

computer plays NIM with the user. Try to make sure that the computer will always win. Make sure that the user does not cheat by taking an illegal number of matches. Make the program as friendly as possible.

The main program submitted was:

```
140 CLEAR
150 EXEC INSTRUCTIONS
160 EXEC NUMSTART
170 EXEC FIRSTPLAYER
180 WHILE REMAINDER <> 1 DO
190    EXEC COMCHOICE
200    IF REMAINDER <> 1 THEN
210       EXEC PLAYCHOICE
220    ENDIF
230 ENDWHILE
240 EXEC WINNER
250 END
```

The procedures were then defined as follows:

INSTRUCTIONS	: Displays information on the game for the user.
NUMSTART	: Allows the user to choose the number of matches for the game.
FIRSTPLAYER	: Gives the user the option of making the first move. If the user chooses to go first then PLAYCHOICE is called to allow the user to do so.
COMCHOICE	: Calculates how many matches the computer should take and reduces REMAINDER accordingly.
PLAYCHOICE	: Allows the user to choose 1, 2 or 3 matches and checks that the number chosen is valid. Reduces REMAINDER accordingly.
WINNER	: Announces the winner of the game.

This is a very good Top-Down design and allows the overall solution to be understood by just reading the main program. The

only part lacking clarity is FIRSTPLAYER. This procedure may or may not call PLAYCHOICE depending on whether the user requests the first move. It would have been clearer to make this fully explicit in the main program. The routines to generate the computer's choice and to accept and check the user's choice, though quite difficult, were handled very well. The screen presentation was also good, although there was no attempt made to represent the matches graphically.

One of the easier problems was a simulation of the game of DODO. The problem definition was as follows:

> The game of DODO is played with two special dice. The blue die has seven sides numbered with the first seven prime numbers (2, 3, 5, 7, 11, 13, 17) and the red die has nine sides numbered with the first nine Fibonacci numbers (1, 1, 2, 3, 5, 8, 13, 21, 34). One player rolls the blue die and the other rolls the red die. The highest score wins. Write a program which, by simulating 1000 games of DODO, decides which die has the better chance of winning.

The students attempting this problem had some difficulty in finding a way to generate the required numbers. When given the hint of storing the numbers in data, they proceeded quite quickly to a good solution. Their main program was:

```
300 REDWINS := 0
310 BLUEWINS := 0
320 COUNT := 1
```

```
330 WHILE COUNT <= 1000 DO
340   EXEC REDNUM
350   EXEC BLUENUM
360   IF REDNUM > BLUENUM THEN
370      REDWINS := REDWINS + 1
380   ELIF  BLUENUM > REDNUM THEN
390      BLUEWINS := BLUEWINS + 1
400   ENDIF
410   COUNT := COUNT + 1
420 ENDWHILE
430 IF REDWINS > BLUEWINS THEN
440    PRINT "red die has better chance"
450 ELIF BLUEWINS > REDWINS THEN
460    PRINT "blue die has better chance"
470 ELIF REDWINS = BLUEWINS THEN
480    PRINT "both have equal chance"
490 ENDIF
500 END
```

The procedure definitions were:

REDNUM : Generates a number for the red die.

BLUENUM : Generates a number for the blue die.

The die numbers in each case were stored in data statements preceded by an appropriate LABEL (RED or BLUE). The procedure REDNUM restores the data pointer to the LABEL RED, generates a random number (X) between one and seven, and then reads through the data list until it comes to the Xth. item. This value is then assigned to REDNUM. A similar procedure is used to generate BLUENUM. In solving this problem the students showed good judgement in the way they used procedures. To have broken the main program into further procedures would have been pointless in such a short and simple program, but to have left the routines in REDNUM and BLUENUM in the main program would have distracted from the clear presentation of the overall

solution. This program (DICEPROJ.CSB) is on the accompanying disk.

These are good examples of both the simpler and more difficult projects undertaken. Further examples are on the accompanying disk. The quality of the work done on these projects is a good indication of the high level of achievement of those students who undertook the course.

8.13 THE USE OF STRUCTURE DIAGRAMS IN MATHEMATICS

A brief experiment was carried out to test if the structure diagrams and the Top-Down method could be applied to other areas of the curriculum. The topic chosen was Leaving Certificate co-ordinate geometry.

It had been observed, over a number of years, that very many students had great difficulty with co-ordinate geometry problems. This was true even when they had been very well drilled in the use of all the relevant formulae (slope, mid-point etc.). It was found that, as long as students were given problems in which only one formula was required, they could manage quite well (e.g. given the slope of a line and the co-ordinates of one point on it, find the equation). However, once the problems became even slightly more difficult, most students were unable to cope. The normal procedure carried out when such problems were being assigned was to discuss the problem and the proposed solution in some

detail with the students. They were then allowed to work alone or in small groups. It was observed that many students who were initially able to discuss the problems intelligently became completely confused when they actually started writing. Typically, students would start off correctly but then either use an inappropriate formula or else simply give up because they were 'lost'. It was felt that they were so involved with the calculations that they were in danger of losing whatever insight they originally had into the solution of the problem.

The class consisted of 23 girls all aged either 15 or 16. All had taken the Intermediate Certificate in 1984 and their maths grades (all lower course) were as follows:

B	5
C	13
D	5

The average number of honours and passes per student were two and five respectively. This indicates quite a poor level of academic achievement and, in addition, they were noted for their generally unruly behaviour and their lack of interest in 'mathematical' subjects. The average attendance over the period of instruction was 88%.

It was felt that the use of structure diagrams would be beneficial in these circumstances. The idea was that students would learn to develop a complete solution to each problem, and set this down in a structure diagram, before doing any calculations. The calculations specified in the diagram would then be carried out and the result of each calculation would be written on the diagram. The student could then see at a

glance how each result fitted into the overall solution. The student would also be 'prompted' by the diagram to carry out the next step in the solution. In short, the student would devise an algorithm and would then simply have to implement the algorithm to get the required result.

EXAMPLE

Given two points a(3,-1) and b(4,5) find the equation of the line through b which is perpendicular to a (see figure 8.5).

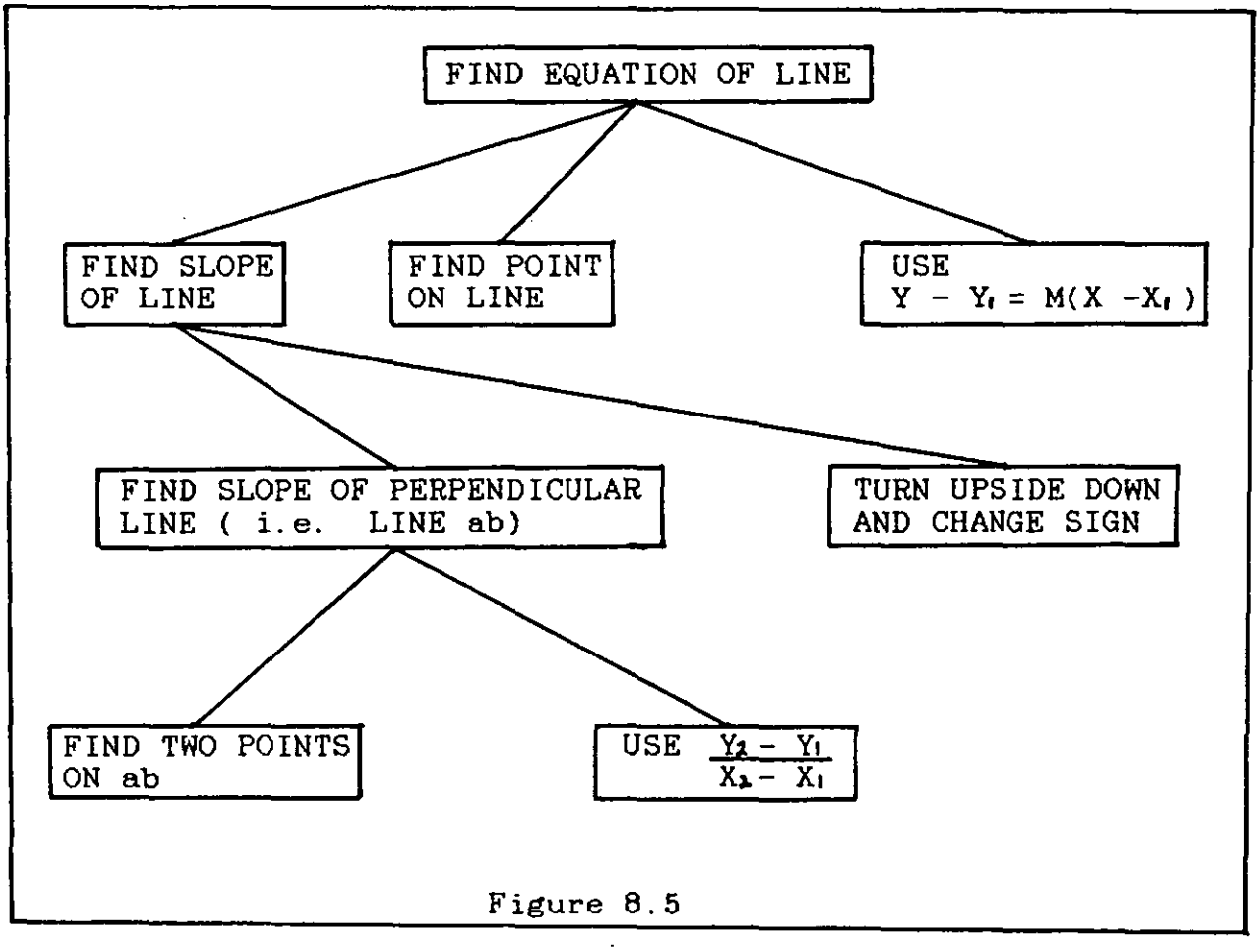


Figure 8.5

Once the diagram is complete, the answer may be found by working up from the bottom until the slope and a point on the line are found and then using the formula $Y - Y_1 = M(X - X_1)$.

The students had been introduced to all the formulae in the

normal way and had done many examples in which only one formula was required. The instruction with the diagrams took place during eight classes, each of forty minutes duration, over a period of two weeks. A ninth class was devoted to a test. At the first class the difficulty of approaching large problems was discussed. It was suggested that they might be broken down into sections, as described in chapter 3 (page 30). At this stage sheets were given to each student containing all the required formulae in structure diagram form (appendix D, pages 235, 236). Further sheets containing sample solutions were also distributed and discussions of how the diagrams were developed took place. When this was understood, they were shown how to find answers from the diagrams.

Throughout the period of instruction many problems were solved, both in the classroom and as homework assignments. The reaction of the students was very favourable and all learned the method quickly. It was noticed that, for the first time, students were distinguishing between finding the solution and finding the answer. They also began to regard finding the solution as the more important of the two. After three days they were given problems to solve, without being explicitly told to use diagrams, but all continued to use them throughout the whole period. The general approach to each problem was to consider the problem statement and then to derive a solution, through a class discussion, on the board. Most members of the class took an interest in, and contributed to, these deliberations. The students would then copy the solution into their notebooks and use this to find the answer. On some

occasions they were left completely to themselves. In the more difficult cases geometrical diagrams were also used to clarify the problem.

At the end of the period a 30 minute test consisting of three problems was given. The problems given were typical of those that had been used during the instruction period. Twenty students took the test and their performance on each problem is discussed below.

QUESTION 1 (figure 8.6).

Find the equation of the line containing the points a(-2,-2) and b(4,6).

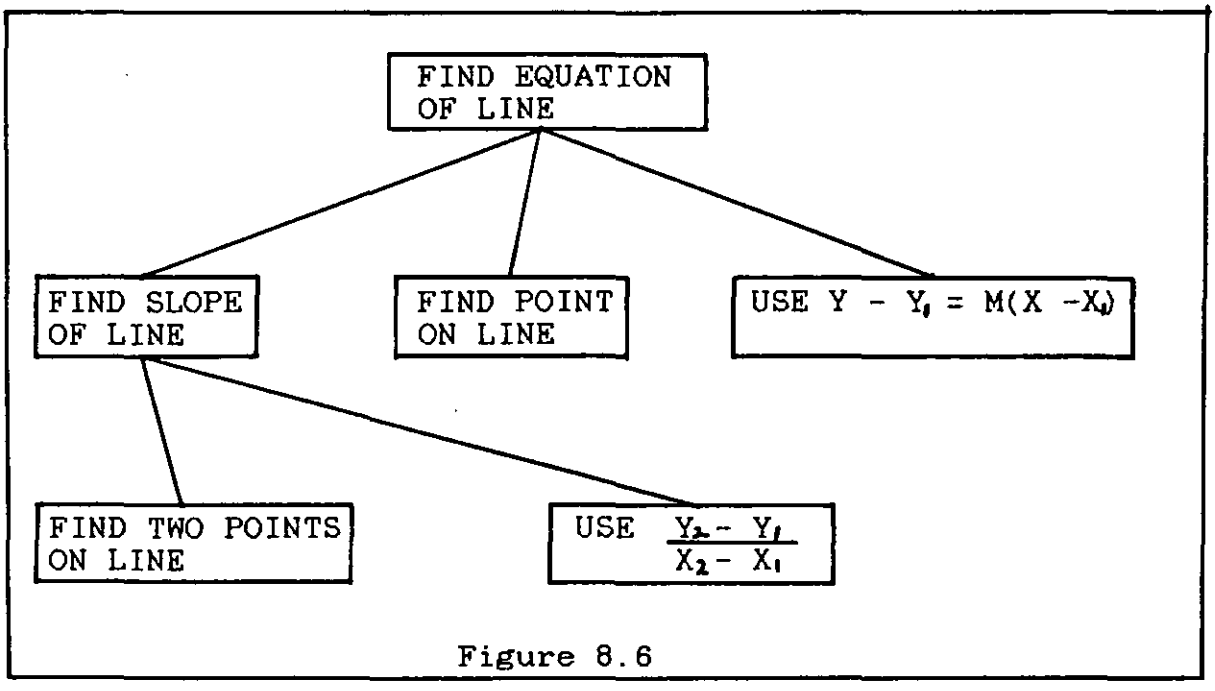


Figure 8.6

Only one student failed to find the complete solution to this problem, although the student concerned did find the first level solution. Of the other nineteen, twelve got the correct answer but all executed the algorithm correctly. (All of their errors were arithmetical.)

Question 2 (figure 8.7).

Find the equation of the line through the point (2,0) which is parallel to the line $4x + 3y - 5 = 0$.

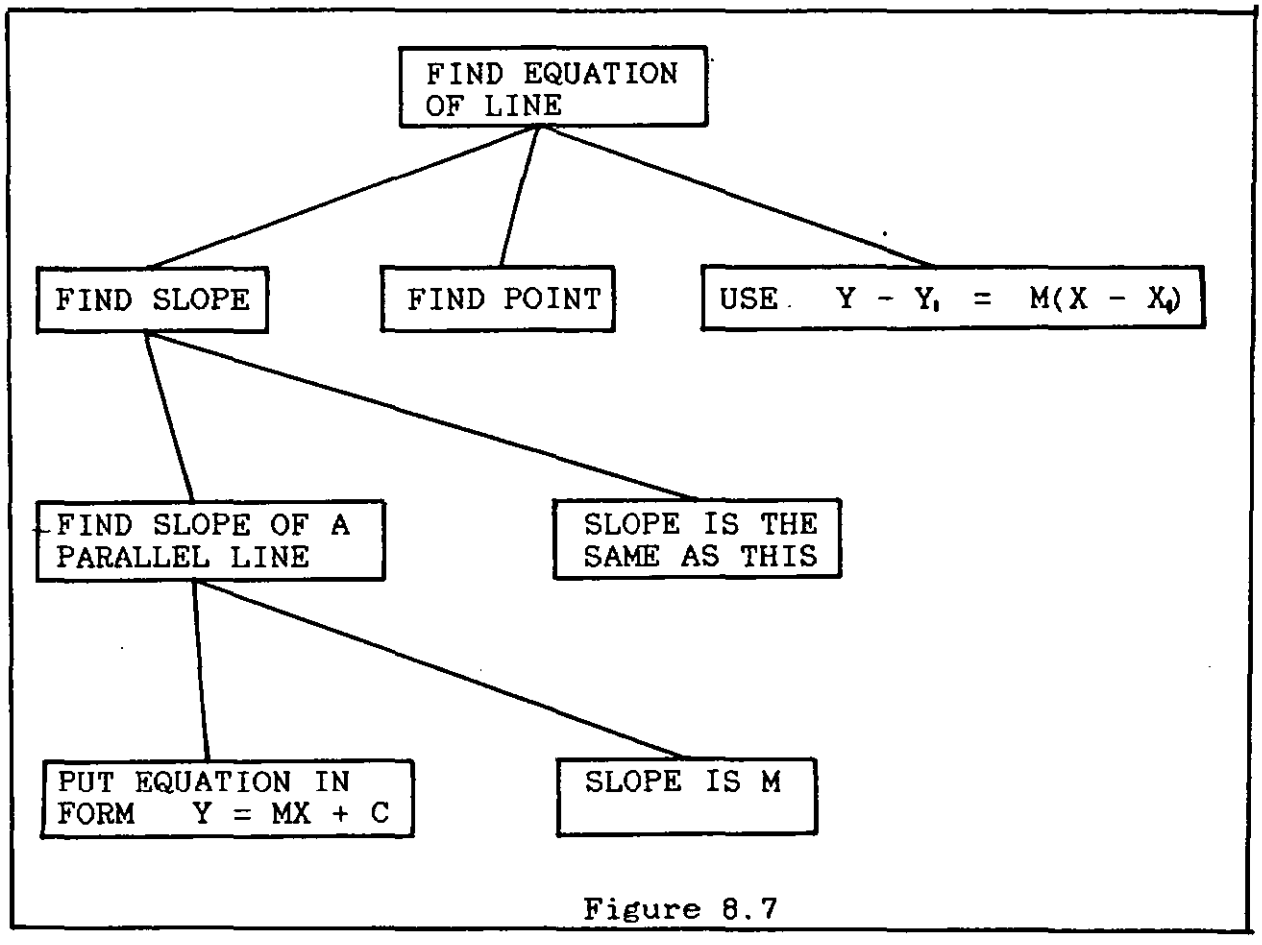


Figure 8.7

Again, only one student failed to find the correct solution (the same student). This time, however, only six students found the correct answer. Of the other thirteen, ten found an incorrect slope for the line $4x + 3y - 5 = 0$ and the other three did not attempt to find the answer.

QUESTION 3 (figure 8.8).

Find the equation of the perpendicular bisector of the line segment joining a(0,0) and b(4,6)

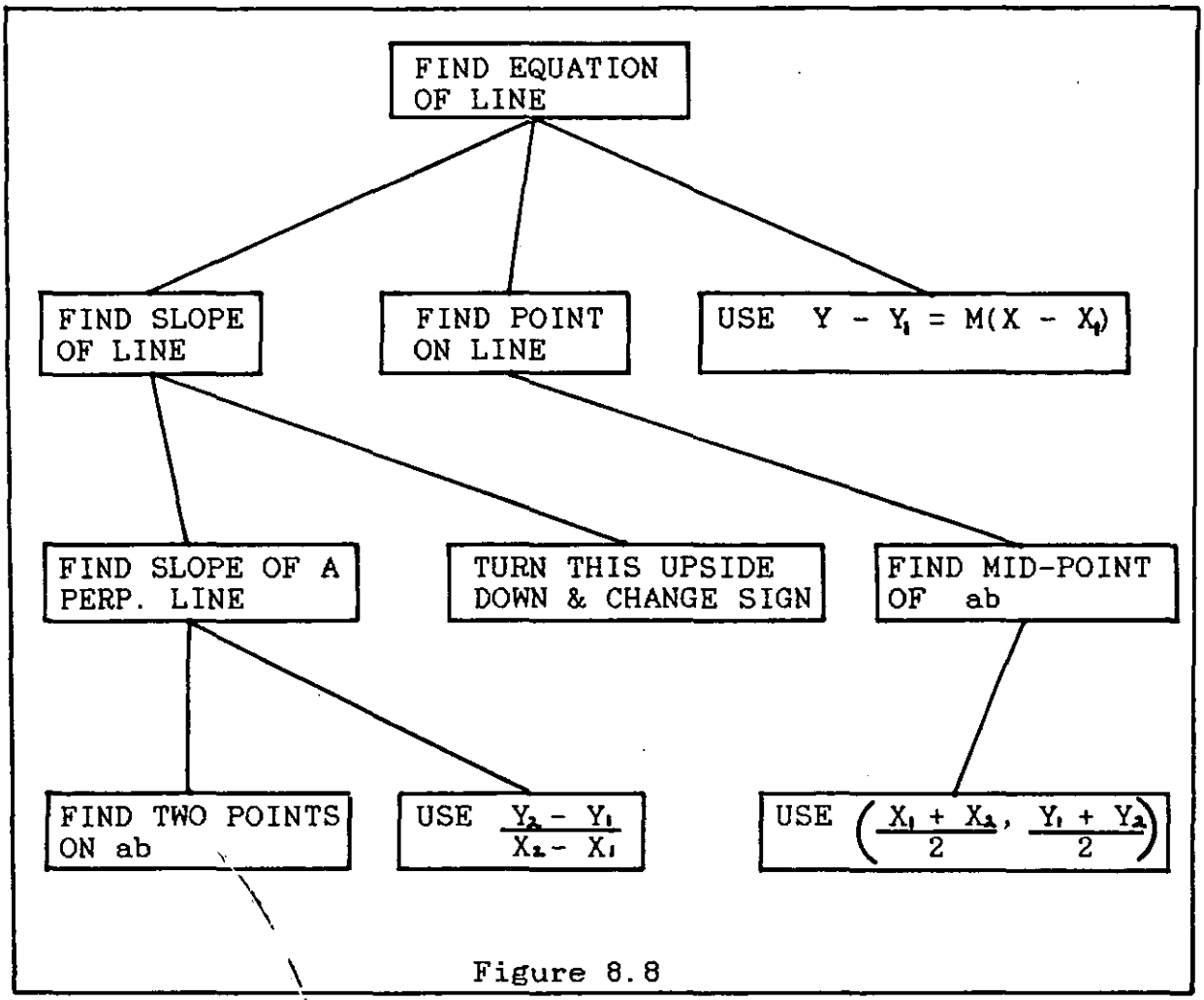


Figure 8.8

In this case eleven students had the correct solution. Of the remaining nine, seven had the algorithm for finding the slope correct but made the mistake of assuming that one of the given points (a or b) was on the required line. The other two were unable to get to grips with the problem at all. Of the eleven with the correct solution, seven got the correct answer.

One encouraging finding which emerged from the test was that all the students were able to execute their algorithms correctly in every case. All the errors arose from arithmetical slips rather than from 'getting lost' in the problems.

It was clear that the students liked using the diagrams and that they regarded them as being helpful. It was encouraging to hear these students discuss solutions among themselves as they worked. This was in sharp contrast to the normal type of conversation that takes place while they work, which normally concerns only the answer. All the students were able to understand the TOP-DOWN nature of the diagrams and were able to use them to find solutions. They were all able to interpret the diagrams. This was evidenced by the fact that they were able to carry out the steps of their solutions in the correct order in every case.

It is felt that the diagrams made a significant contribution towards overcoming their original difficulty and also gave them a tool for finding solutions to more complex problems.

CHAPTER 9

CONCLUSIONS

There is no doubt that the course described here was much more satisfactory than the previous 'traditional' course that had been given to a similar group of students. If it is borne in mind that the total amount of class time involved (approximately 35 hours) was less than what would normally be allotted to Mathematics in one term, and that none of the participants had any previous experience of computing whatsoever, then it is obvious that a considerable amount was accomplished. Both the teachers and the students involved expressed satisfaction with the course and other teachers at the school also noted improvements in the students' ability to argue constructively and to analyse problems.

All the students involved in the course became proficient in the use of a computer system and they were all able to work confidently at the machines without teacher supervision. They became familiar with disk management methods and were able to load and save programs and to use library procedures. Although some were very apprehensive at first, all came to enjoy using the computers. They developed a very positive attitude towards the computers, to the extent that they were prepared to spend large amounts of their free time working at them. Free, unsupervised access to the machines was important in this regard, as the students preferred to work in small groups without having a teacher present. Such extra-curricular work

was very unusual for these students and was one of the most encouraging outcomes of the course.

The mini-language which was designed for the course proved to be totally satisfactory. Elegant, Top-Down solutions to every problem encountered were found using the chosen structures. The similarity between the two control structures, in that they were both governed by boolean expressions, was helpful; and the emphasis placed on the construction of boolean expressions was well justified. The general rejection of ELSE branches and the preference of the students for 'explicit' expressions indicated that the need for clarity was well appreciated by them. The systematic application of the same structures to numerous different problems had the desired effect and the students developed a consistent style of programming. They generally wrote similar algorithms in similar ways, unlike students who are exposed to a complete programming language. The latter tend to use a whole range of structures in an unpredictable way. For example, when constructing fixed iteration loops, such students may use a FOR/NEXT loop at one point and then use a REPEAT/UNTIL loop for an exactly similar situation in another part of the same program. This mixture of structures inhibits the development of programming style and also makes programs difficult to read.

The fact that relatively few solutions submitted differed radically from what was expected also indicated that the students had developed a consistent style. This implies that the teaching was effective and that the students were able to apply what they knew to new situations. This was in marked

contrast to previous experience where students were exposed to a full programming language in a less structured learning environment. In these latter circumstances, methods of approach to problems varied drastically and were very often inappropriate.

The structure diagrams which were designed to represent the mini-language were also considered to be highly successful. They were capable of representing the structures of the language both conveniently and clearly and provided a good introduction to the idea of Top-Down analysis. They were also very easily translated into COMAL; and this process became so 'automatic' for the students that they were able to type code into the computer directly from their diagrams. While the students found it difficult to adapt to the Top-Down approach initially, they gradually came to appreciate its value and used it consistently in the later stages of the course, in preference to direct coding in COMAL. The diagrams were a major factor in impressing on them the need to plan solutions carefully before attempting to consider details or to write any code. The use of the diagrams in a short mathematics module also produced the same effect.

It was difficult to impress on the students the need to test solutions before actually writing any code. They were encouraged to 'trace' all variables and boolean expressions in their programs to check that the required result was produced. This idea was not received too favourably but was carried out in many cases, especially with more difficult problems and with the

projects. In cases where it was done, many students reported that their programs worked correctly on the first run.

The idea of making one's work clearly understandable and communicable was not generally appreciated by the students initially. The consistent use of the diagrams, in which the emphasis was always on achieving clarity and good organisation, eventually produced a change in their approach. The suggested coding style, which emphasised visual layout and the use of meaningful variable names, was very well received and most of their COMAL programs were excellent in this respect. The students therefore learned the importance of organising their work and of presenting it clearly. This was confirmed by other members of the staff who noticed improvements of this nature in the students' other subjects.

The topics chosen for the course were also considered to be satisfactory and the order of presentation of the material worked very well. The problems used to illustrate and reinforce the various concepts were generally satisfactory and were mostly set at the correct level of difficulty for the students involved. Much was learned from the first implementation of the course in this regard, and many problems had to be rejected or modified for the second year. The most important lesson learned here was that the problems had to be on topics with which the students were already quite familiar. They also had to be very carefully graded to ensure that they were within the students' capabilities, while still providing a challenge.

The projects undertaken by the students were executed extremely well. Some of these problems were very difficult but in most cases excellent solutions were achieved. The organisational principles suggested were generally adhered to and the coding style in most of them was excellent. The fact that problems such as these could be tackled successfully was a major achievement, as experience with previous courses would have suggested that these were beyond the reach of such students. Previous experience had shown that students usually either gave up very easily or else resorted to 'hacking' when faced with any moderately difficult problem. However, the combination of mini-language, structure diagrams and translation rules, together with consistency of approach, gave the students the tools they needed to allow them to tackle large problems and to persevere with them until a good solution had been achieved.

The success of the structure diagrams in co-ordinate geometry suggests that the Top-Down method could be applied to other areas of the curriculum and there is a need for future experimentation in this field. There are many topics in Mathematics and Science which could certainly be treated this way and there may also be the possibility of application in subjects such as Business Studies, Geography etc., especially with weaker students. Any problem which may be broken down into a set of procedures may be treated in this way and the insight gained into the hierarchical nature of those procedures may be of great value. In the context of Computer Science, further research is required to establish if the method can be used with more complex problems and with students from both older and

younger age groups. There is also a need to test if the method works as conveniently with other implementation languages such as BASIC, PASCAL and LOGO. The translation rules for BASIC are more complex than those for COMAL and it would be interesting to see if the translation process would become 'automatic' for students using BASIC. It would also be interesting to see how students who had already taken a 'traditional' course might react when exposed to this approach.

In conclusion, it is felt that the course was an unqualified success as the primary aims of promoting computer literacy and problem-solving skills were achieved. All the students gained an insight into the process of Top-Down programming and became familiar with the use of a computer. Many of them decided that they would like to pursue careers in computing as a result of the enjoyment they had derived from the course. However, even if they do not become computer specialists, it is very likely that most of them will encounter computers in their working lives and the experience gained from the course is certain to be of assistance to them in this regard.

APPENDICES

AND

REFERENCES

APPENDIX A

This appendix contains samples of class notes distributed to the students.

As COMAL is not supplied with the machine it must be loaded into memory before it is used. The language is supplied on disk. The procedure for loading it into memory is as follows:

1. Put COMAL disk into drive 1 and switch on.
2. A> appears on the screen.
3. Type COMAL-80 and press 'return'.
4. COMAL message now appears and you are asked if you require 'error texts'. Type Y (for 'YES').

Error texts are messages that are printed on the screen when you make a mistake. If you typed N (for 'NO') then you would merely get an error number when you made a mistake and you would have to look up the manual to find what the mistake was.

5. * now appears on the screen. This is the COMAL prompt to let you know that the machine is ready to accept COMAL programs.

IMMEDIATE EXECUTION MODE

In immediate execution mode the commands that you give to the machine are carried out immediately after you press 'return'. These are called DIRECT COMMANDS. For example if you type CLEAR the screen will be cleared immediately. Some of the more important direct commands are given below:

LIST ....This causes the commands of whatever program is in memory to be written on the screen.
NEW ....This erases the current program from memory
RUN ....This causes the program in memory to be executed
CLEAR ...This clears all text from the screen but has no effect on the program in memory. (CLEAR may also be used in deferred execution mode).

DEFERRED EXECUTION MODE (PROGRAMS)

A program is a sequence of deferred commands i.e. a list of instructions that are stored in the computer's memory. They are executed when the direct command RUN is given. Programs may be written to solve a huge variety of problems i.e. to create a computer game, to calculate tax, to teach a geography lesson, to test a student's knowledge etc. The fact that so many different problems can be solved using the same computer accounts for their great power.

In deferred execution mode each command (or statement) is preceded by a number.

When the command is typed in it is stored in memory. When the direct command RUN is given the commands in memory are carried out in sequence from the lowest numbered to the highest. It is normal to number the commands in 'steps' of 10. It is unusual to type in a whole program without making some errors. The process of correcting errors is called EDITING.

SIMPLE EDITING IN APPLE COMAL-80

A. ERROR IS DETECTED BEFORE YOU PRESS 'RETURN':
   1. Use backarrow to position cursor over incorrect character.
   2. Type in correct character.
   3. When satisfied press 'return'.

B. ERROR NOTICED AFTER YOU PRESS 'RETURN':
   1. If error is a syntax error then the faulty line will be displayed with the cursor over (or near) the error. Proceed as in A above.
   2. Other errors may be corrected by retyping the whole line.
   3. To remove a line completely from a program type DEL (for delete) followed by the appropriate line number. You must leave a space between the word DEL and the number e.g. DEL 50.
   4. If you get 'stuck' press the escape (ESC) button on the keyboard.

THE PRINT STATEMENT

The PRINT statement is used to write on the screen (or on the printer...later). To write something on the screen you simply type:

```
PRINT "anything you like in here"
```

Note that the words to be written are enclosed in double quotation marks. You may use the command PRINT with nothing after it to print a blank line on the screen. The use of blank lines to separate text helps to make the screen more readable.

SAMPLE PROGRAM:
To clear screen and then write name and address.

```
10 CLEAR
20 PRINT
30 PRINT
40 PRINT "M.BRADY"
50 PRINT " HOLY FAITH CONVENT"
60 PRINT "  THE COOMBE"
70 PRINT "   DUBLIN.8."
80 END
```

EXERCISE

WRITE PROGRAMS TO CLEAR THE SCREEN AND THEN PRINT:

1. Your own name and address.
2. Your own name and address indented (as on an envelope).
3. Your own name and address in top r.h. corner of screen (as on a letter)
4. The numbers 1 to 5 on successive screen lines.
5. The numbers 1 to 5 with blank lines in between.
6. The numbers 1 to 5 in a diagonal across the screen.
7. The numbers 1 to 5 on the same line with spaces between them.
8. A 'solid' rectangle made of asterisks
9. An 'empty' rectangle made of asterisks.
10. A 'solid' rectangle in the centre of the screen.

STRINGS

A string is a sequence of characters enclosed in double quotation marks. Characters are letters, digits, spaces, commas etc.

```
  e.g.   "SALLY O'BRIEN"        "THE AREA IS "
         "DONALD DUCK"          etc.
```

STRING VARIABLES

Like NUMERIC variables, STRING variables are also 'boxes' in the computer's memory. To enable the computer to distinguish between the two types of variable, String variable names always end with the dollar sign ($).

VALID VARIABLE NAMES

NUMERIC	STRING
LENGTH	NAME$
INTEREST	ADDRESS$
RATEPERHOUR	ANSWER$

Values are given to STRING variables by the same statements that are used for NUMERIC variables. (i.e. ASSIGNMENT, INPUT AND READ)

ASSIGNMENT

The value that is to be given to the variable must be enclosed in double quotation marks.

```
e.g.  NAMES$ := "MARY LOU"
      YEAR$  := "FIFTH"
      HOBBY$ := "DANCING"
```

INPUT

This is exactly the same as for NUMERIC variables

```
e.g.  INPUT FIRSTNAME$
      INPUT "WHAT GRADE DID YOU GET ":GRADE$
      INPUT "DO YOU WANT TO TRY AGAIN ":REPLY$
```

N.B. No quotes are required when typing in the value during program execution (in response to an INPUT statement).

Once a value has been put into a STRING variable it may then be printed out in the same way as a NUMERIC variable.

```
e.g. 50 INPUT "WHAT IS YOUR NAME ":NAME$
     60 PRINT "HI THERE ";NAME$
```

COMAL contains statements for manipulating string variables. Some of these will be introduced later.

DIMENSIONING STRING VARIABLES

The computer stores STRING values and NUMERIC values in different ways. Before setting up a 'box' for a STRING variable the computer needs to know how much space to set aside...IT NEEDS TO KNOW THE MAXIMUM NUMBER OF CHARACTERS THAT YOU EXPECT TO PUT IN THE BOX.
If the variable is to hold an exam grade then the maximum number of characters would be two. If it was to hold a name then the maximum would be about 25. If it was to hold addresses the maximum might be about 50.
In COMAL you tell the computer how much space to set aside by using a DIMENSION statement:

```
e.g. 10 DIM GRADE$ OF 2
     20 DIM NAME$ OF 25
```

The DIM statement must come BEFORE the variable is given a value. It is a good idea to DIMENSION all your STRING variables together at the start of your program

```
e.g.    10 CLEAR
        20 DIM NAME$ OF 25
        30 DIM YEAR$ OF 6
        40 INPUT "WHAT IS YOUR NAME ":NAME$
        50 PRINT
        60 PRINT "HELLO THERE ";NAME$
        70 INPUT "WHAT YEAR ARE YOU IN ":YEAR$
        80 PRINT
        90 PRINT YEAR$;" YEAR !! HOW EXCITING !!"
```

If you forget to include the DIM statement for a STRING variable then you will get an UNDEFINED VARIABLE error message when you try to give a value to the variable.

If you try to give a longer value to a STRING variable than has been allowed for in the DIM statement then the extra characters will be 'chopped off'. For example if the STRING variable SHOP$ has been DIMENSIONED for 10 characters and you try to give it the value "QUINNSWORTH" which contains 11 characters then this would be shortened to "QUINNSWORT". What would happen if you gave it the value "SUPERQUINN" or "DUNNE'S STORES"

It is O.K. to give a shorter value than has been allowed for in the DIM statement.

EXERCISE

1. Write a program which asks the user to supply her name and age and then makes a suitable comment.
2. Write a program which asks the user for her name and the name of her boyfriend and then makes some comment.

VARIABLES

1. A variable may be thought of as a box in the computer's memory.
2. Each variable has a value and a name.
3. You may create as many variables as you like in a program.
4. A lot of programming consists of manipulating variables.
5. We will be using two different 'types' of variable. These are NUMERIC and STRING.

NUMERIC ... value may be any number (whole,decimal,+ or -)

STRING ....value may be any group of characters

RULES FOR VARIABLE NAMES

1. May be up to 16 characters long
2. Characters must be letters or digits
   i.e. no special characters such as commas, spaces etc.
3. First character must be a letter.
4. Cannot be a COMAL keyword.
5. String variable names end with the dollar sign.

It is important that you use meaningful variable names i.e. try to choose names which suggest what the role of the variable is. For example if a variable is to contain a number which represents the average of some group of numbers then call the variable AVERAGE. A variable may have only one value at any given time. This means that if you change the value of a variable then the old value is destroyed. In COMAL there are three ways in which values may be given to variables:

1. ASSIGNMENT STATEMENTS
2. INPUT STATEMENTS
3. READ STATEMENTS

The READ statement will be dealt with later.

1.ASSIGNMENT STATEMENT

An assignment statement consists of a VARIABLE NAME on L.H.S, the assignment symbol (:=) , and an 'expression' on the R.H.S.

```
SCORE:= 12
WAGES:= HOURS * RATE
   PI:= 3.1416
HOURS:= MINUTES * 60
```

1. The symbol := is read as "becomes equal to"
   e.g. SCORE becomes equal to 12
2. In each case the expression on the R.H.S. is evaluated and the value is given to the variable named on the L.H.S.
3. If the expression on the R.H.S. contains variables then they must have been given values earlier in the program (i.e. they must be INITIALISED). If any variables on the R.H.S have not been initialised then the assignment statement is invalid ,the program will 'crash' and an error message will be displayed.
4. The variables on the R.H.S. are unchanged by the assignment statement.

2. INPUT STATEMENT

INPUT statements allow values to be given to variables ,by the user, while the program is RUNning.

```
e.g.        30  INPUT LENGTH
```

When the program reaches line 30 it stops and a question mark is displayed on the screen. This is a signal to the user that some data is required by the program. Whatever value is typed in is then given to the variable named in the input statement and the program continues on to the next line. Of course a question mark is not a great prompt for the user but fortunately the programmer may display meaningful prompts very simply as follows:

```
e.g.    30 INPUT "what score did you get ":SCORE
        40 INPUT "how old are you":AGE
```

The prompt,which must be in double quotation marks, is displayed instead of the question mark.The variable name is separated from the prompt by a colon.

OUTPUTTING THE VALUE OF A VARIABLE

To output the value of a variable on the screen use PRINT

```
e.g.            PRINT SCORE
```

This causes the value of SCORE to be written on the screen.

NOTE 1.PRINT SCORE is not the same as PRINT "SCORE"
NOTE 2.You may include a 'prompt' in the PRINT statement as in the INPUT statement but use a semicolon to separate the 'prompt' from the variable name.

```
e.g.            PRINT "your present score is ";SCORE
```

EXERCISE INVOLVING INPUT,ASSIGNMENT AND PRINT STATEMENTS

1. Input length and breadth of a rectangle. Output it's area
2. Input principal, rate and time. Output simple interest.
3. Input any whole number. Output the next one.
4. Input any number. Output it's square and it's cube.
5. Input any four numbers. Output their average.
6. Input degrees celsius. Output degrees fahrenheit
   FAHRENHEIT=(9/5)*CELSIUS + 32
7. Input radius of sphere. Output it's volume.
   VOLUME = 4/3(PI * RADIUS^3)
8. Input radius and height of a cylinder. Output it's volume and it's surface area.
9. Input a student's test score and the maximum possible score for the test. Output percentage mark.

When programming these problems on the machine try to make the screen output 'pretty' by clearing the screen at the start,printing blank lines to separate the 'output section' from the 'input section' etc.
In short try to make the screen readable.

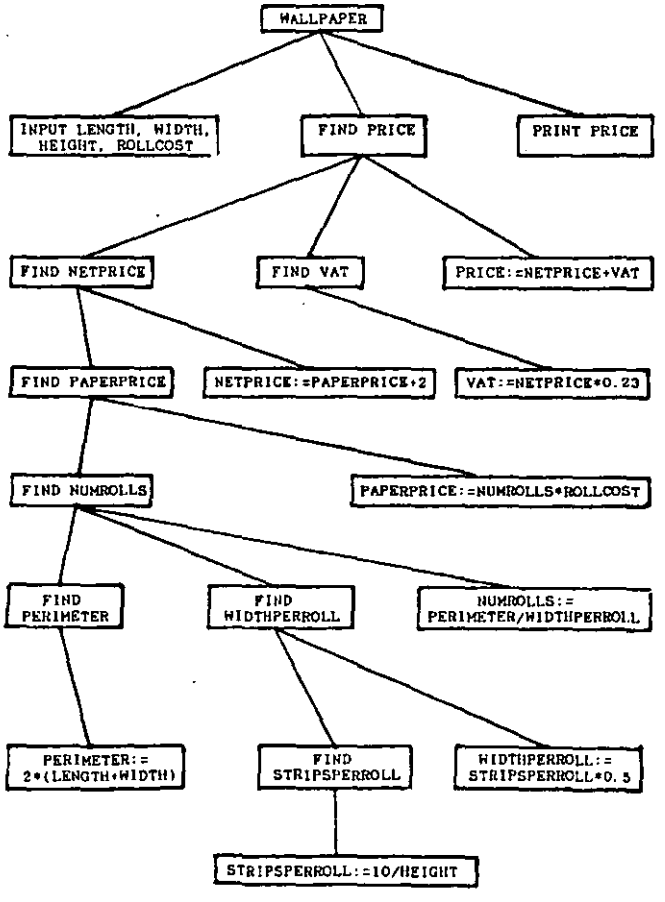
EXERCISES

1. Write a program to calculate the total price of an item if the nett price and the rate of VAT are input.
2. Write a program to calculate nett pay if gross pay, tax free allowance and rate of tax are input.
3. Design a program which could be used to estimate the total cost of laying concrete paths around rectangular gardens (all four sides). The length and width of the garden should be input (in feet) along with the width of the required path. The cost of cement is 20p. per square foot and VAT is charged at 23%.
4. Write a program to estimate the cost of making curtains in which the width and height of the window are input (in feet) along with the cost per yard of the material. The finished curtains should go 6 inches above and below the window. The width of the finished curtains should be twice that of the window. The cost of lining is £2 per yard. There is a fixed charge of £20 per pair of curtains. All material is 48 inches wide. VAT is 23%.

SAMPLE PROBLEM

The wallpaper department of a large retail store wish to give computerised estimates by phone to their customers. The customer supplies the length, width and height of the room to be papered and the price per roll of the paper chosen (from the store's catalogue). This information is then given to the computer and it is expected to estimate the number of rolls required and the total cost of papering the room.

The length of a roll of wallpaper is 10 metres.
The width " " " " " " 0.5 "
Paste costs 2 pounds per room.
VAT is charged at 23% on both paste and paper.
You needn't allow for doors, windows etc.



1. Write a program which calculates the total cost of buying a quantity of some particular item where:
   A. The cost per item
   B. The number of items
   C. The rate of V.A.T.

are all INPUT. Assume that there is a standard charge of $20 for delivery regardless of how many items are bought.

2. Write a program to calculate the time taken (in hours) to heat a swimming pool from 15 degrees centigrade to 25 degrees centrigrade, given that the heating plant can heat 100 cubic metres of water through the required temperature range in 90 minutes. The following items should be INPUT:
   A. Length of pool in metres
   B. Width of pool in metres
   C. Depth of pool in metres

3. Write a program to find the cost of repairing a car where:
   A. The net cost of all materials
   B. The number of hours spent working on car
   C. The cost per hour for labour

are INPUT. The V.A.T. rate on materials is 23% and the V.A.T. on labour is 5%.

4. Write a program to find the cost of ordering pencils and rulers for a school where:
   A. The number of pencils
   B. The number of rulers
   C. The price per pencil
   D. The price per ruler

are INPUT. The rate of V.A.T. on both items is 23%.

5. A journey is to be made by car and boat. The car averages 40 m.p.h. and the boat averages 12 m.p.h. Write a program in which:
   A. The distance to be travelled by car
   B. The distance to be travelled by boat

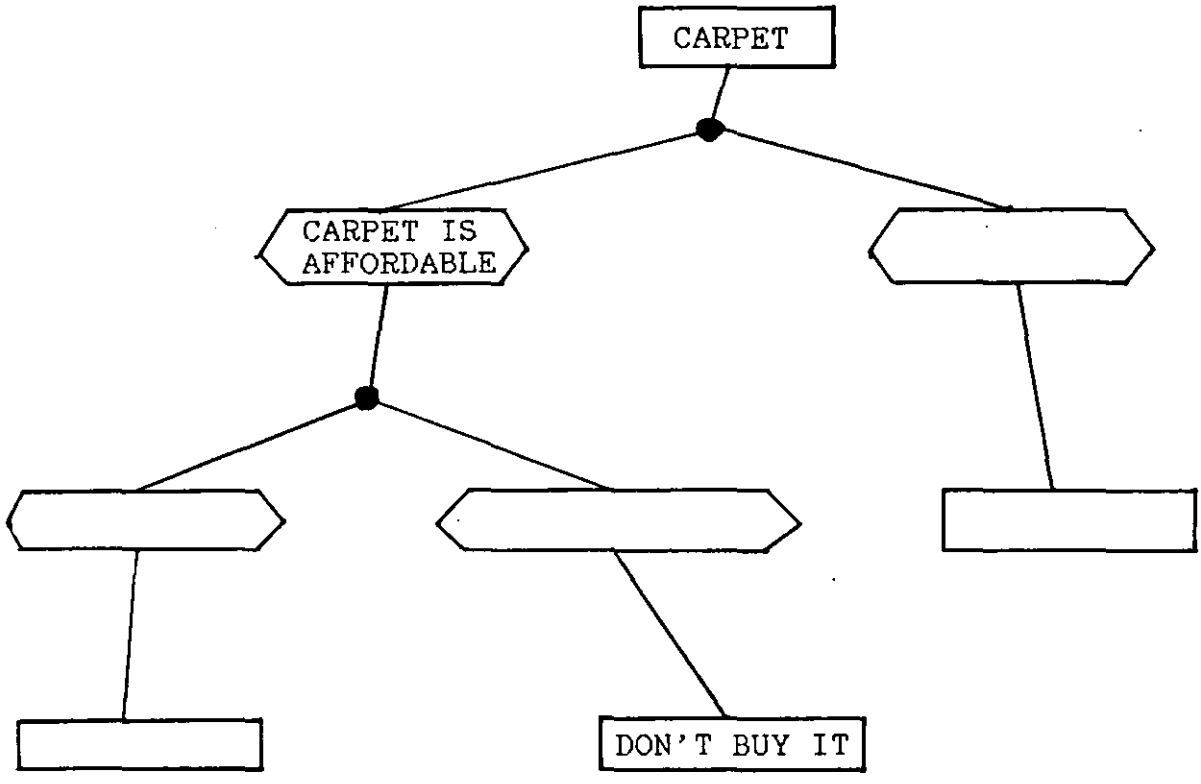
are INPUT and which then calculates the total time taken.

6. A driver has discovered that his car averages 30 m.p.g. in city driving and 40 m.p.g. in country driving. Write a program in which:
   A. The number of miles of city driving per week
   B. The number of miles of country driving per week
   C. The price per gallon of petrol

are INPUT and which then calculates the total cost per week for petrol.

In the following problem you are required to insert the given statements into the correct boxes. (Some of the given statements are not required at all so it is up to you to decide which ones are required and where they should go)

STATEMENT OF PROBLEM
==========
A person wants to buy a carpet. It must be both affordable and of a suitable colour. Draw a diagram to illustrate how a carpet is 'checked'.



STATEMENTS TO BE INSERTED
---

1. DON'T BUY IT
2. CARPET IS BIG ENOUGH
3. CARPET IS NOT A SUITABLE COLOUR
4. BUY IT
5. NEGOTIATE A DISCOUNT
6. CARPET IS 100% WOOL
7. CARPET IS NOT AFFORDABLE
8. PRICE INCLUDES FITTING CHARGE
9. CARPET IS A SUITABLE COLOUR
10. CARPET FITS THE ROOM
11. BUY RUG PROTECTORS FOR THE FURNITURE

PROBLEMS REQUIRING USE OF IF STRUCTURE

1. Write a program in which the user is asked to input any number. The program should then decide if the number is positive, negative or zero and output the result.

2. Input a number. If it is bigger than 5 then multiply it by 2. Otherwise multiply it by 3. Output the result.

3. Input a number and say whether or not it's square is greater than 10000.

4. Input two numbers. If they are different then output the smaller one. If they are the same then print a message saying so.

5. Write a program in which the user is given the choice of calculating either simple interest or compound interest (Principal, Rate and Time to be input).

6. Write a program which offers the user the choice of calculating the area of a rectangle, a triangle or a circle. When the user has indicated which shape is required the program should ask for the relevant measurements, calculate the area and output the result.

7. Write a program in which the computer carries on a conversation with the user. The computer should ask the user some simple questions (i.e. What school do you go to ? Do you like your school ? etc.) and make a suitable comment on each of the user's responses.

8. Write a program to calculate an air fare for the user. The standard fare is £400 but there is a 15% discount for travellers under 21. If a first class seat is required there is an extra charge of £200. This first class surcharge is the same for all travellers regardless of age. The program should prompt the user to supply all the necessary information.

9. Input a number and say if it is an integer (whole number)
   HINT: If X is an integer is it equal to INT(X) ?

10. Input a number and say if it is even.
    HINT: If X is even what can be said about X MOD 2 ?

11. Input a number and say if it is a multiple of 7.
    HINT: This is almost the same as the previous problem

PROBLEMS INVOLVING ARRAYS

1. Fill an Array with N random numbers between 1 and 20 and then:
   A. Print them all out.
   B. Print out the first 10 elements.
   C. Print out the second 10 elements.
   E. Print out every second element. Give the user the choice of printing the odd or the even numbered elements.

2. Fill two Arrays A and B with N elements each. Then create two new Arrays, C and D, from:
   a. The sums of the corresponding elements in A and B.
   b. The larger of the corresponding elements in A and B.

3. Find the sum of all the terms in an Array of N elements.

4. Find the average of the elements in an Array of N elements.

5. Given two arrays, each containing the same number of elements, determine how many pairs of corresponding elements are equal.

6. Read numbers into two Arrays. Print out the Array with the bigger average. If they have the same average then print out a suitable message. (They need not contain the same number of elements).

7. Find the number of zero's in an Array.

8. Set each element of an Array to the value of the sum of all the elements in the Array.

9. Set each element of an Array to the value of itself plus all the preceding elements (i.e. 1,2,3,4 should become 1,3,6,10).

10. Write out the elements of an array in reverse order.

11. Write out the elements of an Array containing N elements in the order First, Last, Second, Second-Last etc.

12. Fill an Array with N numbers and then swap the First with the Last, the Second with the Second Last etc., until the whole Array has been reversed.

13. Find the biggest and smallest numbers in an N element Array.

14. Find the average of all the elements in an array & round the answer off to the nearest whole number. Then change each element of the array into the difference between it and the average (i.e. if and element was 2 below the average it should be changed to -2).

15. Test if an Array is Palindromic (i.e. the same when read from either end.).

16. Output the first N lines of Pascal's Triangle.

PROBLEMS ON STRING MANIPULATION

1. Input a string. Output the first two characters.

2. Input a name. Check if the surname begins with O'. (i.e. O'BRIEN, O'DOWD etc.)

3. Input a string. Output the last character.

4. Input a string. Output the string backwards.

5. Input a string. Output the middle character (or the two middle characters if there is an even number of characters).

6. Input a string. Say how many vowels are contained in it.

7. Input a name (Christian name and Surname). Ouput the Christian name only.

8. Input a name (Christian name and Surname). Output the Surname only.

9. Input a name (Christian name and Surname). Output the initials only.

10. Input a name (Christian name and Surname) into a string variable. Output the name with the Surname first.

11. Input a string. Say if it is a Palindrome.

12. Write a program to create 'spoonerisms'. A spoonerism is got by swapping the first letters of two words i.e. JOE SOAP would become SOE JOAP.
The correct names should be in DATA statements. The program should read these and output the adjusted names. The end of the data should be marked with the word "END".

13. Write a program to check if words containing the letters 'i' and 'e' are spelt correctly. The rule is " i before e except after c ". You will need to check that the words don't contain the letters 'cie' and that they don't contain the letters 'ei' immediately after a 'c'. The words should be input at the keyboard.

PROBLEMS ON STRING ARRAYS

1. Read a list of 10 names from data into an array. Print them out again in reverse order.

2. Read the names and scores of 10 pupils from data lines into two Arrays. Print out the name of the pupil with the highest score and the name of the pupil with the lowest score. Assume that all the scores are different.

3. Read 10 names and scores. Find the average (mean) score and print out the names of those who scored above average.

4. Read 10 names and scores from data. Print out each name saying whether that persons score was above , below or equal to the average. (The average should be rounded off to the nearest whole number.)

5. Read 10 names from data. (Each name consists of a christian name and a surname , separated by a space.) Print out the initials of each person.

6. Read 20 names into an array. Each name should consist of a title (MR. MRS. etc.), a christian name and a surname. Write programs to:

   a. Print out all the mens' names.
   b. Print out all the women's names.
   c. Print out all the names in the order:
      surname, christian name, title
   d. Print out all the names in the order:
      title, initial, surname

7. Read 20 names into an array. Each name consists of a christian name and a surname. Create a new array in which the same names are stored with the surname first. Give the user the choice of having the names output in either order.

8. Write a program to print out the words of the song "THE 12 DAYS OF CHRISTMAS". The 'gifts' for each day should be held in data statements.

```
DATA "FIRST","A PARTRIDGE IN A PEAR TREE"
DATA "SECOND"," 2 TURTLE DOVES AND A "
DATA "THIRD"," 3 FRENCH HENS"  etc.
```

APPENDIX B

This appendix contains listings of two COMAL demonstration programs, FONEDEMO and QUIZDEMO, that were given to the students.

```
0010 // M.BRADY
0020 //
0030 // 27/2/1984
0040 //
0050 // PROGRAM TO DEMONSTRATE USE OF PROCEDURES
0060 //
0070 ////////////////////////////////////////////////////////////
0080 // PROGRAM TO ALLOW USER TO FIND A
0090 // PHONE NUMBER BY INPUTTING A
0100 // NAME AND VICE-VERSA.
0110 // ALL INPUT IS ASSUMED TO BE VALID
0120 //
0130 ////////////////////////////////////////////////////////////
0140 // INITIALISATION SECTION
0150 //
0160 DIM TESTNAME$ OF 20
0170 DIM NAME$ OF 20
0180 //
0190 ////////////////////////////////////////////////////////////
0200 // MAIN PROGRAM
0210 //
0220 EXEC MENU
0230 WHILE CHOICE=1 OR CHOICE=2 DO
0240  IF CHOICE=1 THEN
0250   EXEC FINDNUM
0260  ELIF CHOICE=2 THEN
0270   EXEC FINDNAME
0280  ENDIF
0290  EXEC MENU
0300 ENDWHILE
0310 EXEC GOODBYE
0320 END
0330 //
0340 // END OF MAIN PROGRAM
0350 ////////////////////////////////////////////////////////////
0360 PROC MENU
0370  //
0380  // PROCEDURE TO PRINT THE OPTIONS
0390  // ON THE SCREEN AND ACCEPT
0400  // THE USER'S INPUT.
0410  //
0420  CLEAR
0430  PRINT TAB(10);"TELEPHONE PROGRAM"
0440  PRINT TAB(10);"================"
0450  PRINT
0460  PRINT
0470  PRINT "YOU MAY :"
0480  PRINT
0490  PRINT "     1. SEARCH FOR A NUMBER"
0500  PRINT
0510  PRINT "     2. SEARCH FOR A NAME"
0520  PRINT
0530  PRINT "     3. QUIT"
0540  CURSOR 1, 22
0550  INPUT "TYPE IN NUMBER OF YOUR SELECTION  ": CHOICE
0560 ENDPROC MENU
0570 //
0580 ////////////////////////////////////////////////////////////
0590 PROC FINDNUM
0600  //
0610  // PROCEDURE TO ASK FOR A NAME
0620  // AND THEN TO SEARCH DATA FOR
0630  // THE APPROPRIATE NUMBER
```

```
0640  //
0650  CLEAR
0660  PRINT
0670  PRINT
0680  INPUT "ENTER NAME...": TESTNAME$
0690  READ NAME$, NUMBER
0700  WHILE NOT (NAME$=TESTNAME$ OR NAME$="END") DO
0710   READ NAME$, NUMBER
0720  ENDWHILE
0730  IF NAME$=TESTNAME$ AND NAME$<>"END" THEN
0740   PRINT
0750   PRINT
0760   PRINT "PHONE NUMBER OF ";TESTNAME$;" IS ";NUMBER
0770  ELIF NAME$="END" THEN
0780   PRINT
0790   PRINT
0800   PRINT "THIS NAME IS NOT IN THE DIRECTORY"
0810  ENDIF
0820  RESTORE
0830  EXEC SPACEBAR
0840 ENDPROC FINDNUM
0850 //
0860 ////////////////////////////////////
0870 PROC FINDNAME
0880  //
0890  // PROCEDURE TO ASK FOR A NUMBER
0900  // AND THEN TO SEARCH DATA FOR
0910  // APPROPRIATE NAME.
0920  CLEAR
0930  PRINT
0940  PRINT
0950  INPUT "ENTER NUMBER.......": TESTNUMBER
0960  READ NAME$, NUMBER
0970  WHILE NOT (NUMBER=TESTNUMBER OR NAME$="END") DO
0980   READ NAME$, NUMBER
0990  ENDWHILE
1000  IF NUMBER=TESTNUMBER AND NUMBER<>0 THEN
1010   PRINT
1020   PRINT
1030   PRINT
1040   PRINT "THE PERSON WITH THIS PHONE NUMBER IS ";NAME$
1050  ELIF NAME$="END" THEN
1060   PRINT
1070   PRINT
1080   PRINT
1090   PRINT "THIS NUMBER IS NOT IN THE DIRECTORY"
1100  ENDIF
1110  RESTORE
1120  EXEC SPACEBAR
1130 ENDPROC FINDNAME
1140 //
1150 ////////////////////////////////////
1160 PROC GOODBYE
1170  //
1180  // PROCEDURE TO CLEAR SCREEN
1190  // AND GIVE 'END' MESSAGE
1200  //
1210  CLEAR
1220  CURSOR 10, 10
1230  PRINT "END OF PROGRAM"
1240  CURSOR 10, 11
1250  PRINT "=============="
1260 ENDPROC GOODBYE
1270 //
1280 ////////////////////////////////////////////////////
1290 PROC SPACEBAR
1300  //
```

```
1310  // PROCEDURE TO HALT PROGRAM
1320  // EXECUTION UNTIL THE SPACEBAR
1330  // IS HIT.
1340  //
1350  // THIS PROCEDURES CONTAINS
1360  // STATEMENTS THAT HAVE NOT BEEN
1370  // EXPLAINED IN CLASS.
1380  //
1390  CURSOR 1, 24
1400  PRINT "HIT SPACE BAR TO CONTINUE ";
1410  POKE 256, 0
1420  REPEAT
1430  UNTIL PEEK(256)=32
1440 ENDPROC SPACEBAR
1450 //
1460 ////////////////////////////////////////////////////////////////
1470 // DATA SECTION
1480 //
1490 DATA "PAT", 1
1500 DATA "JOE", 2
1510 DATA "MARY", 3
1520 DATA "KATE", 4
1530 DATA "END", 0
```

```
0010 // MULTIPLE CHOICE QUIZ
0020 //
0030 // M.BRADY 5/3/1984
0040 //
0050 // TO ILLUSTRATE SIMPLE
0060 // ERROR CHECKING ROUTINE
0070 // AND USE OF INVERSE,FLASH
0080 // NORMAL AND BELL PROCEDURES
0090 //
0100 ////////////////////////////////
0110 // INITIALISATION SECTION
0120 //
0130 DIM QUESTION$ OF 50
0140 DIM OPTION1$ OF 20
0150 DIM OPTION2$ OF 20
0160 DIM OPTION3$ OF 20
0170 DIM OPTION4$ OF 20
0180 DIM CORRECT$ OF 1
0190 DIM RESPONSE$ OF 1
0200 SCORE:=0
0210 //
0220 ////////////////////////////////
0230 // MAIN PROGRAM
0240 //
0250 EXEC INSTRUCTIONS
0260 COUNT:=1
0270 WHILE COUNT<=3 DO
0280  EXEC READER
0290  EXEC DISPLAY
0300  EXEC ANSWER
0310  COUNT:=COUNT+1
0320 ENDWHILE
0330 EXEC GOODBYE
0340 END
0350 //
0360 ////////////////////////////////
0370 PROC INSTRUCTIONS
0380  //
0390  // PROCEDURE TO INFORM USER ON
0400  // USE OF PROGRAM
0410  //
0420  CLEAR
0430  PRINT TAB(14);"QUIZ PROGRAM"
0440  PRINT TAB(14);"============"
0450  EXEC BELL
0460  CURSOR 1, 8
0470  PRINT "THIS IS A QUIZ IN WHICH YOU WILL BE"
0480  PRINT "ASKED 3 QUESTIONS"
0490  PRINT
0500  PRINT "FOR EACH QUESTION YOU WILL BE GIVEN"
0510  PRINT "FOUR OPTIONS A,B,C AND D."
0520  PRINT
0530  PRINT
0540  PRINT "TYPE A ,B ,C OR D IN RESPONSE TO EACH"
0550  PRINT "QUESTION AND THEN PRESS RETURN"
0560  EXEC SPACEBAR
0570 ENDPROC INSTRUCTIONS
0580 //
0590 ////////////////////////////////
0600 //
0610 PROC READER
0620  //
0630  // PROCEDURE TO READ IN A
0640  // QUESTION & THE FOUR
0650  // POSSIBLE ANSWERS FROM
0660  // THE DATA LINES.
```

```
0670  //
0680  READ QUESTION$
0690  READ OPTION1$
0700  READ OPTION2$
0710  READ OPTION3$
0720  READ OPTION4$
0730  READ CORRECT$
0740 ENDPROC READER
0750 //
0760 ////////////////////////////////
0770 //
0780 PROC DISPLAY
0790  //
0800  // PROCEDURE TO DISPLAY THE
0810  // QUESTION & POSSIBLE ANSWERS
0820  // ON THE SCREEN.
0830  //
0840  CLEAR
0850  PRINT "QUESTION NO...";COUNT;"              SCORE = ";SCORE
0860  PRINT "========================================"
0870  PRINT
0880  PRINT
0890  PRINT QUESTION$
0900  PRINT
0910  PRINT
0920  PRINT
0930  PRINT "A. ";OPTION1$
0940  PRINT
0950  PRINT "B. ";OPTION2$
0960  PRINT
0970  PRINT "C. ";OPTION3$
0980  PRINT
0990  PRINT "D. ";OPTION4$
1000 ENDPROC DISPLAY
1010 //
1020 ////////////////////////////
1030 //
1040 PROC ANSWER
1050  //
1060  // PROCEDURE TO ASK FOR A
1070  // RESPONSE FROM THE USER
1080  // & TO CHECK THAT IT IS
1090  // A VALID RESPONSE.
1100  //
1110  CURSOR 1, 22
1120  EXEC INVERSE
1130  PRINT "ENTER CHOICE...(A,B,C OR D) ";
1140  EXEC NORMAL
1150  INPUT " ": RESPONSE$
1160  WHILE NOT (RESPONSE$="A" OR RESPONSE$="B" OR RESPONSE$="C" OR
      RESPONSE$=" D") DO
1170   CURSOR 1, 22
1175   EXEC NORMAL
1180   EXEC FLASH
1190   EXEC BELL
1200   PRINT "MUST BE A,B,C OR D          ";
1210   EXEC NORMAL
1220   INPUT "...": RESPONSE$
1230  ENDWHILE
1240  IF RESPONSE$=CORRECT$ THEN SCORE:=SCORE+1
1250 ENDPROC ANSWER
1260 //
1270 ////////////////////////////////
1280 ////////////////////////////
```

```
1290 PROC INVERSE
1300  //
1310  // PROCEDURE TO GIVE SCREEN
1320  // DISPLAY IN INVERSE MODE
1330  //
1340  POKE 61490.0, 63
1350 ENDPROC INVERSE
1360 //
1370 ////////////////////////////
1380 ////////////////////////////
1390 PROC FLASH
1400  //
1410  // PROCEDURE TO GIVE SCREEN
1420  // DISPLAY IN FLASH MODE
1430  //
1440  POKE 61490.0, 127
1450 ENDPROC FLASH
1460 //
1470 ////////////////////////////
1480 ////////////////////////////
1490 PROC NORMAL
1500  //
1510  // PROCEDURE TO GIVE SCREEN
1520  // DISPLAY IN NORMAL MODE
1530  //
1540  POKE 61490.0, 255
1550 ENDPROC NORMAL
1560 //
1570 ////////////////////////////
1580 //
1590 PROC GOODBYE
1600  //
1610  // PROCEDURE TO PRINT OUT THE
1620  // SCORE AND GIVE END OF
1630  // PROGRAM MESSAGE.
1640  //
1650  CLEAR
1660  CURSOR 5, 10
1670  PRINT "YOU GOT ";SCORE;" RIGHT OUT OF 3"
1680  CURSOR 1, 22
1690  PRINT "PROGRAM EXECUTION FINISHED"
1700 ENDPROC GOODBYE
1710 //
1720 ////////////////////////////
1730 //
1740 PROC BELL CLOSED
1750  //
1760  // PROCEDURE TO SOUND BELL
1770  // FIVE TIMES.
1780  //
1790  COUNT:=1
1800  WHILE COUNT<=5 DO
1810   POKE 61509.0, 135
1820   POKE 62416.0, 217
1830   POKE 62417.0, 251
1840   CALL 56126.0
1850   COUNT:=COUNT+1
1860  ENDWHILE
1870  //
1880 ENDPROC BELL
1890 //
1900 ////////////////////////////
1910 //
1920 PROC SPACEBAR
1930  //
```

```
1940  // PROCEDURE TO HALT PROGRAM
1950  // EXECUTION UNTIL SPACEBAR
1960  // IS HIT.
1970  //
1980  CURSOR 1, 24
1990  PRINT "HIT SPACE BAR TO CONTINUE ";
2000  POKE 256, 0
2010  REPEAT
2020  UNTIL PEEK(256)=32
2030 ENDPROC SPACEBAR
2040 //
2050 ////////////////////////////////
2060 //
2070 // DATA SECTION
2080 //
2090 DATA "THE FIRST PRESIDENT OF IRELAND WAS "
2100 DATA "DE VALERA", "DOUGLAS HYDE"
2110 DATA "CHARLIE HAUGHEY", "GARRET FITZGERALD", "B"
2120 //
2130 DATA "WHO INVENTED THE TELEPHONE "
2140 DATA "SEAN DOHERTY", "MICHAEL NOONAN"
2150 DATA "ALEXANDER BELL", "EINSTEIN", "C"
2160 //
2170 DATA "WHO WROTE WAR AND PEACE"
2180 DATA "TOLSTOY", "DICKENS"
2190 DATA "JOHN LENNON", "RONALD REAGAN", "A"
```

APPENDIX C

This appendix contains the problems that were assigned as end of year projects and a sample of one of the projects submitted.

PROBLEM No. 1

Computers are often used to produce cheques. As well as writing the amount of money in numeric form it is also necessary to write it in words:
e.g. IR£234.56 should be written as:
TWO HUNDRED AND THIRTY FOUR POUNDS FIFTY SIX PENCE
Write a program which will convert amounts of money input in numeric form into words. The program should terminate when a value of zero is input.

PROBLEM No. 2

Write a currency conversion program which allows the user to convert from IR£ to:

A. POUNDS STERLING
B. FRENCH FRANCS
C. U.S. DOLLARS
D. CANADIAN DOLLARS

and from any of these back to IR£. These four options should be displayed in the main menu. When this choice is made the user should be asked to input the current rate i.e. the number of dollars (or francs etc.) that are equivalent to IR£1. The user should then be presented with another menu:

1. CONVERT PUNTS TO DOLLARS (OR FRANCS etc.)
2. CONVERT DOLLARS (OR FRANCS etc.) TO PUNTS.
3. RETURN TO THE MAIN MENU.

When this selection is made the user should be allowed to do as many conversions of that type as she requires before being returned to this menu.

PROBLEM No. 3

The game of NIM is normally played with matchsticks. Starting with any number of matches each of two players is allowed to remove 1,2 or 3 matches at a time. Whoever is left with the last match loses. The 'trick' is to leave your opponent with 5 matches in which case no matter how many she takes you can always ensure that she is left with the last one. To make sure that she is left with 5 you should ensure that she is left with 9 (13,17,21,25,29...).
Write a program in which the computer plays NIM with the user. Try to make sure that the computer will win. Make sure that the user doesn't cheat by taking an illegal number of matches.
Make the program as friendly as possible.

PROBLEM No. 4

Write a program to give a test in arithmetic. The user should be given the choice of ADDITION, SUBTRACTION, MULTIPLICATION OR DIVISION and should be asked 10 questions on the chosen topic.
The questions should be generated randomly and if the answer is correct she should be given 10 marks. If incorrect she should be given a second attempt and if this is correct she should be given 5 marks. If the second attempt is also incorrect then no marks should be given but the correct answer should be displayed before going on to the next question.
A final score should be displayed before returning to the menu.

PROBLEM No. 5

Write a program which determines on which day of the week any date this century falls. The date should be input in numeric form.

PROBLEM No. 6

The game of DODO is played with two special dice. The blue die has seven sides numbered with the first seven prime numbers (2,3,5,7,11,13,17) and the red die has nine sides numbered with the first nine FIBONACCI numbers (1,1,2,3,5,8,13,21,34). One player rolls the red die and the other rolls the blue die. The highest score is the winner.
Write a program which simulates 1000 throws of the two dice to find out which has the better chance of winning.

PROBLEM NO. 7

Write a program which will pick a random integer between 1 and 100 (incl.). The user is asked to guess the number. For each try the user should be told if the guess is too high or too low until the correct number is input. The program should then output the number of guesses that were required. If this is less than 7 then a congratulatory message should also be output. Otherwise output an appropriate insult.

PROBLEM No. 8

Write a program in which the user picks a random integer between 1 and 100 (incl.) and the computer then has to guess what the number is. When the computer makes a guess the user should be asked to input either S (too small), B (too big) or C (correct).
the program should then output the number of guesses taken.

PROBLEM No .9

Write a program which will generate 10 random five word sentences from 'dictionaries' of nouns, verbs, adjectives and adverbs. Each sentence should be of the form:

The (adjective) (noun) (verb) (adverb).

e.g. The big computer worked efficiently.

PROBLEM No. 10

Write a program which prints out a calendar for any month where the user inputs the name of the month, the number of days in the month and the day of the week that the 1st. of the month falls on. The output should be of the form:

FEBRUARY

MON	TUES	WED	THURS	FRI	SAT	SUN
					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28		

```
0010 // PROJECT BY VALERIE TRAYNOR
0020 // AND IRENE COMERFORD
0030 //
0040 // SUBMITTED AT END OF FIRST
0050 // YEAR COURSE  MAY 1984
0060 //
0070 ////////////////////////////////
0080 ////////////////////////////////
0090 // DIM SECTION
0100 DIM TURN$ OF 10
0110 DIM ANSWER$ OF 10
0120 ////////////////////////////////
0130 // MAIN PROGRAM
0140 CLEAR
0150 EXEC INSTRUCTIONS
0160 EXEC NUMSTART
0170 EXEC FIRSTPLAYER
0180 WHILE REMAINDER<>1 DO
0190  EXEC COMCHOICE
0200  IF REMAINDER<>1 THEN
0210   EXEC PLAYCHOICE
0220  ENDIF
0230 ENDWHILE
0240 EXEC WINNER
0250 END
0260 //
0270 //END MAINPROGRAM
0280 ////////////////////////////////
0290 //
0300 PROC INSTRUCTIONS
0310  //THIS PROCEDURE WILL TELL THE
0320  //USER HOW TO PLAY THE GAME NIM
0330  //
0340  PRINT "******************************"
0350  PRINT
0360  PRINT "            NIM "
0370  PRINT "           =====  "
0380  PRINT
0390  EXEC BELL
0400  PRINT
0410  PRINT
0420  PRINT "THE GAME NIM IS PLAYED WITH "
0430  PRINT "MATCHSTICKS .STARTING WITH "
0440  PRINT "ANY NUMBER OF MATCHES YOU CHOOSE "
0450  PRINT "EACH PLAYER IS ALLOWED TO REMOVE "
0460  PRINT "1,2 OR 3 MATCHES AT A TIME . "
0470  PRINT
0480  PRINT
0490  PRINT "WHOEVER IS LEFT WITH THE LAST MATCH          LOSES "
0500  //
0510  EXEC SPACEBAR
0520 ENDPROC INSTRUCTIONS
0530 //
0540 ////////////////////////////////
0550 //
0560 PROC NUMSTART
0570  //THIS PROCEDURE ASKES THE USER
0580  //TO INPUT THE NUMBER OF MATCHES
0590  //WHICH HE WANTS THE GAME TO
0600  //BEGIN WITH
0610  //
0620  CLEAR                                                    230
0630  PRINT "TYPE IN THE NUMBER OF MATCHES "
```

```
0640  PRINT
0650  PRINT "WHICH YOU WANT TO BEGIN WITH "
0660  PRINT
0670  PRINT "IT MUST BE MORE THAN 5 "
0680  CURSOR 1, 10
0690  INPUT "    ": NUMSTART
0700  PRINT
0710  PRINT
0720  WHILE NOT (NUMSTART>5) DO
0730   EXEC FLASH
0740   EXEC BELL
0750   PRINT "MUST BE GREATER THAN 5..."
0760   EXEC NORMAL
0770   INPUT "...": NUMSTART
0780  ENDWHILE
0790  REMAINDER:=NUMSTART
0800  //
0810 ENDPROC NUMSTART
0820 //
0830 ////////////////////////////
0840 //
0850 PROC FIRSTPLAYER
0860  //
0870  //THIS PROCEDURE WILL GIVE THE USER
0880  //THE CHOICE OF GOING FIST OR SECOND
0890  //
0900  CLEAR
0910  PRINT
0920  PRINT "////////////////////////////////"
0930  CURSOR 1, 6
0940  PRINT "DO YOU WANT TO GO FIRST OR SECOND ?  "
0950  PRINT
0960  PRINT
0970  PRINT
0980  PRINT
0990  INPUT "    ": ANSWER$
1000  IF ANSWER$="FIRST" THEN
1010   EXEC PLAYCHOICE
1020  ENDIF
1030  EXEC SPACEBAR
1040 ENDPROC FIRSTPLAYER
1050 //
1060 ////////////////////////////
1070 //
1080 PROC COMCHOICE
1090  //THIS PROCEDURE WILL DECIDE WHAT
1100  //NUMBER THE COMPUTER SHOULD TAKE
1110  //
1120  CLEAR
1130  PRINT "COMPUTER CHOICE"
1140  PRINT "==============="
1150  NUM:=(REMAINDER-5) MOD 4
1160  RANDOM
1170  IF NUM=0 THEN
1180   NUM:=RND(1,3)
1190  ENDIF
1200  PRINT NUM
1210  REMAINDER:=REMAINDER-NUM
1220  TURN$:="COMPUTER "
1230  PRINT
1240  PRINT "THERE ARE  ";REMAINDER;"REMAINING"
1250  EXEC SPACEBAR
1260  //
1270 ENDPROC COMCHOICE
1280 //
1290 ////////////////////////////
```

```
1300 //
1310 PROC PLAYCHOICE
1320  //THIS PROCEDURE WILL ASK THE USER
1330  //TO TYPE IN A NUMBER OF HIS CHOICE
1340  //AND TO CHECK THAT IT IS A VALID
1350  //ONE
1360  CLEAR
1370  PRINT "PLAYER CHOICE"
1380  PRINT "=============="
1390  PRINT
1400  PRINT
1410  PRINT
1420  PRINT "TYPE IN THE NUMBER YOU WANT "
1430  PRINT
1440  PRINT
1450  PRINT
1460  PRINT
1470  INPUT "    ": NUM
1480  PRINT
1490  WHILE NOT (NUM=1 OR NUM=2 OR NUM=3) DO
1500   CURSOR 1, 22
1510   EXEC FLASH
1520   EXEC BELL
1530   PRINT "MUST BE 1,2 OR 3    ";
1540   EXEC NORMAL
1550   INPUT "...": NUM
1560  ENDWHILE
1570  REMAINDER:=REMAINDER-NUM
1580  TURN$:="YOU "
1590  PRINT
1600  PRINT "THERE ARE ";REMAINDER;"REMAINING"
1610  PRINT
1620  PRINT
1630  PRINT
1640  EXEC SPACEBAR
1650  //
1660 ENDPROC PLAYCHOICE
1670 //
1680 ////////////////////////////
1690 //
1700 PROC WINNER
1710  //THIS PROCEDURE WILL PRINT THE
1720  //WINNER
1730  PRINT
1740  CLEAR
1750  PRINT "****************************************"
1760  PRINT
1770  CURSOR 1, 5
1780  IF TURN$="YOU " THEN
1790   PRINT "CONGRATULATIONS !!!! YOU HAVE WON "
1800  ELIF TURN$="COMPUTER " THEN
1810   PRINT "HARD LUCK !! YOU HAVE BEEN BEATEN "
1820  ENDIF
1830  PRINT
1840  CURSOR 1, 10
1850  PRINT "****************************************"
1860 ENDPROC WINNER
1870 ////////////////////////////
1880 ////////////////////////////
1890 PROC INVERSE
1900  //
1910  // PROCEDURE TO GIVE SCREEN
1920  // DISPLAY IN INVERSE MODE
1930  //
1940  POKE 61490.0, 63
1950 ENDPROC INVERSE
```

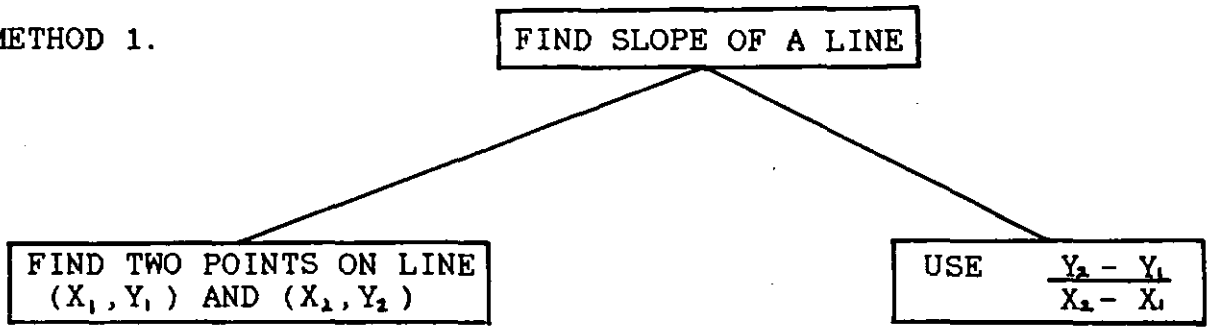
```
1960 //
1970 ////////////////////////////
1980 ////////////////////////////////
1990 //
2000 PROC BELL
2010  //
2020  // PROCEDURE TO SOUND BELL
2030  // NUMBER TIMES.
2040  //
2050  COUNT:=1
2060  NUMBER:=5
2070  WHILE COUNT<=NUMBER DO
2080   POKE 61509.0, 135
2090   POKE 62416.0, 217
2100   POKE 62417.0, 251
2110   CALL 56126.0
2120   COUNT:=COUNT+1
2130  ENDWHILE
2140  //
2150 ENDPROC BELL
2160 //
2170 ////////////////////////////////////
2180 ////////////////////////////
2190 PROC FLASH
2200  //
2210  // PROCEDURE TO GIVE SCREEN
2220  // DISPLAY IN FLASH MODE
2230  //
2240  POKE 61490.0, 127
2250 ENDPROC FLASH
2260 //
2270 ////////////////////////////
2280 ////////////////////////////
2290 PROC NORMAL
2300  //
2310  // PROCEDURE TO GIVE SCREEN
2320  // DISPLAY IN NORMAL MODE
2330  //
2340  POKE 61490.0, 255
2350 ENDPROC NORMAL
2360 //
2370 ////////////////////////////
2380 PROC SPACEBAR
2390  //
2400  // PROCEDURE TO HALT PROGRAM
2410  // EXECUTION UNTIL SPACEBAR
2420  // IS HIT.
2430  //
2440  CURSOR 1, 24
2450  PRINT "HIT SPACE BAR TO CONTINUE ";
2460  POKE 256, 0
2470  REPEAT
2480  UNTIL PEEK(256)=32
2490 ENDPROC SPACEBAR
2500 //
2510 ////////////////////////////////
```

APPENDIX D

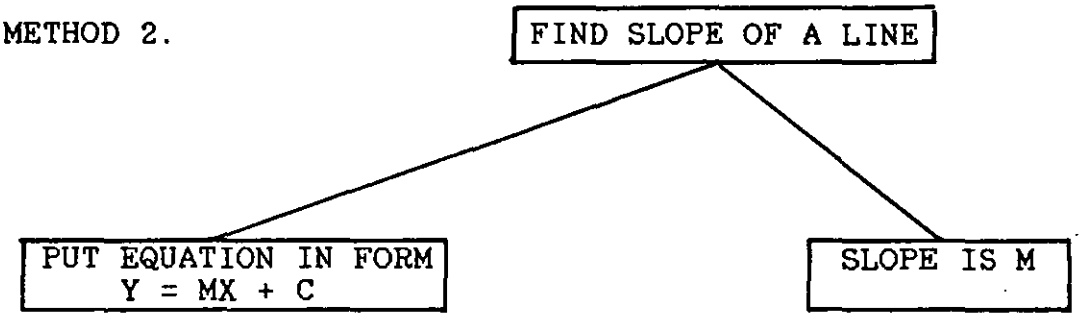
This appendix contains the Mathematics class notes that were distributed to the students who undertook the Mathematics module.

SLOPE OF A LINE
==========

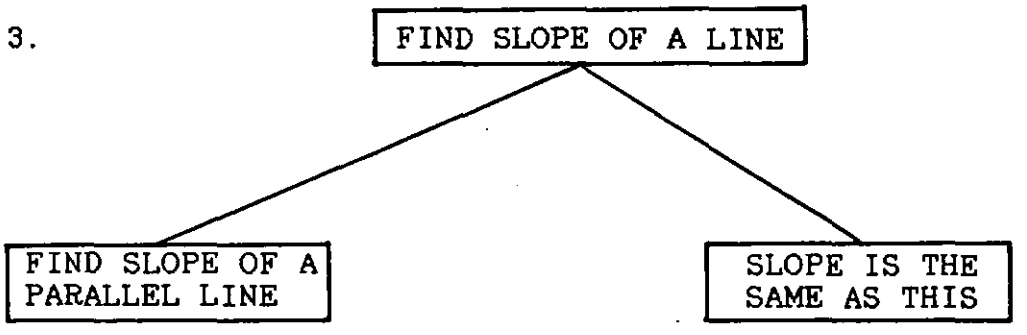
METHOD 1.



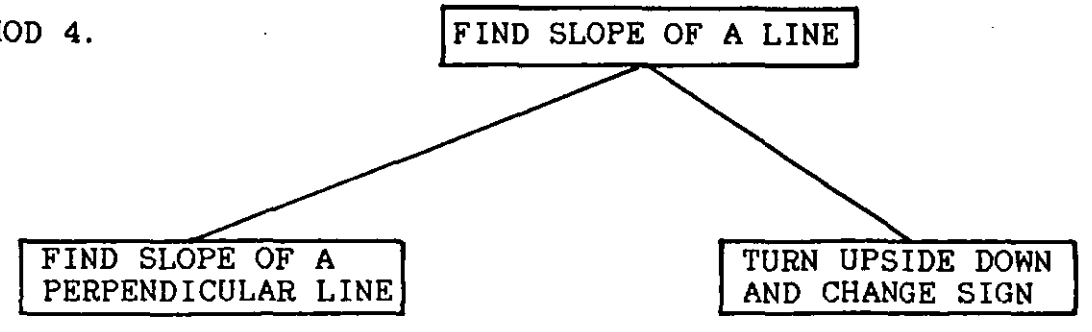
METHOD 2.



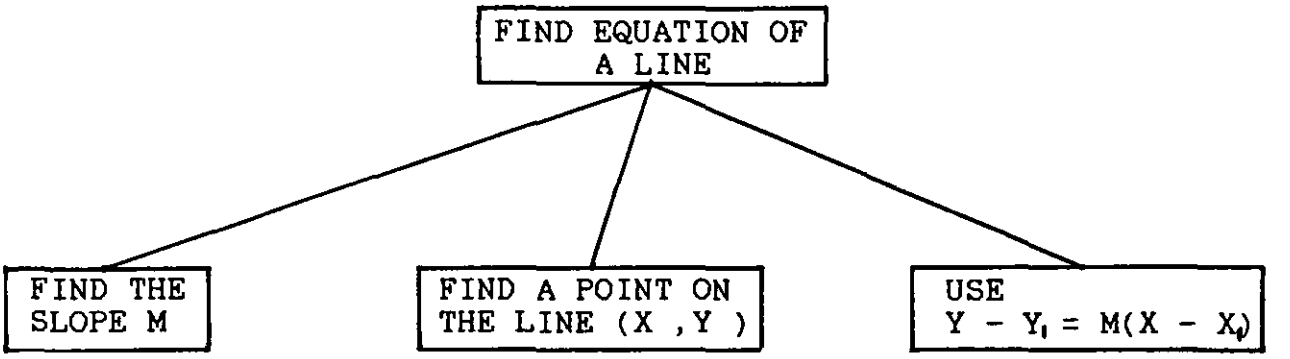
METHOD 3.



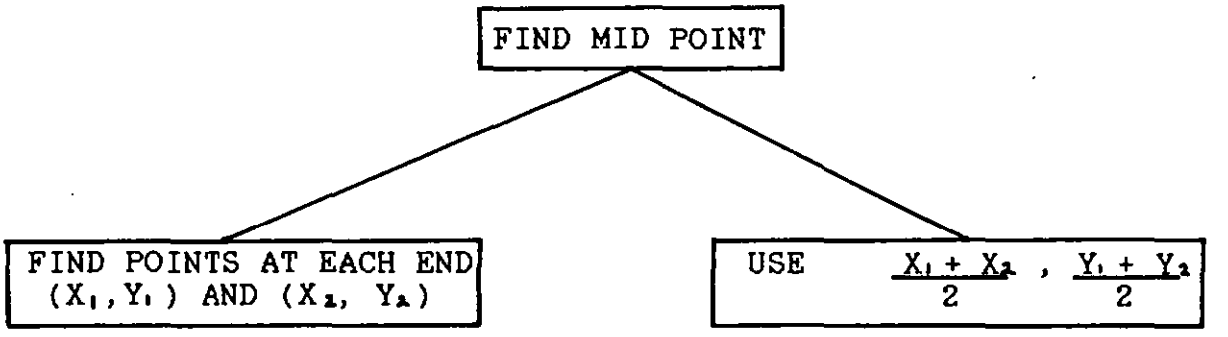
METHOD 4.



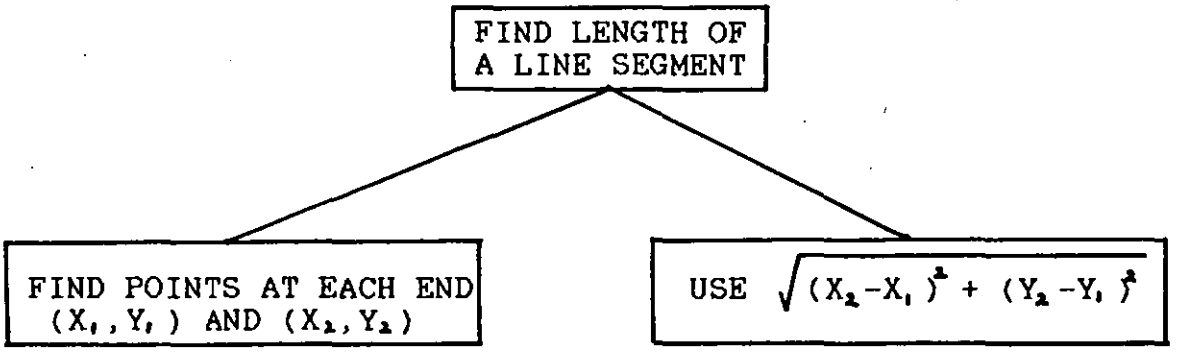
EQUATION OF A LINE



MID POINT OF A LINE SEGMENT



LENGTH OF A LINE SEGMENT



APPENDIX E

This appendix contains instructions for using the accompanying disk.

The disk contains both the METANIC COMAL-80 language and sample programs referred to in the thesis.

System requirements : Apple II+ with 64K memory and fitted with a Z-80 card.

Starting up : When disk is booted CP/M is loaded. When the CP/M prompt (A >) appears, type COMAL-80. You will then be asked if error texts are required. If you reply YES (Y) then error messages will be given when appropriate. If you reply NO (N) then error numbers only will be given. It is normal to reply YES. The COMAL prompt (*) then appears.

Loading Programs : When the COMAL-80 language has been loaded you may catalog the disk by typing CAT. To load a program simply type LOAD followed by the program name. Program names are followed by the suffix .CSB in the catalog but it is not necessary to include the suffix when loading a program.

Programs on the disk :

STAT.COM	FORMAT.COM
NIMPROJ.CSB	COMAL-80.COM
PIP.COM	CASHPROJ.CSB
FONEDEMO.CSB	QUIZDEMO.CSB
NUM8PROJ.CSB	NUM7PROJ.CSB
DATEPROJ.CSB	DICEPROJ.CSB
SUMSPROJ.CSB	CHEMQUIZ.CSB
MENUEXER.CSB	QUIZEXER.CSB
QUIZEX2.CSB	COMPOUND.CSB

REFERENCES

[1] SMITH J., WINNING I. (1985). Computers in Schools. Limerick, National Institute for Higher Education.

[2] Department of Education (1984). Programme for Action in Education 1984 - 1987. Dublin, Stationery Office.

[3] Curriculum and Examinations Board (1984). Issues and Structures in Education. Dublin, Curriculum and Examinations Board.

[4] PAPERT S. (1980). Mindstorms. Brighton, Harvester Press.

[5] Department of Education (1985). Computer Studies Module - Junior Cycle. Dublin, Department of Education.

[6] Department of Education (1984). Rules and Programme for Secondary Schools. Dublin, Stationery Office.

[7] Department of Education Advisory Committee on the Use of Computers at Second Level (1984). The Design and Use of Structured Algorithms. Dublin, Department of Education.

[8] Hativa N. (1984). Good Teaching of Mathematics as Perceived by Undergraduate Students. INTERNATIONAL JOURNAL OF MATHEMATICAL EDUCATION IN SCIENCE AND TECHNOLOGY, 15, 5, 605-615.

[9] WELLS M.B.(1980). Reflections on the Evolution of Algorithmic Language. In METROPOLIS N., HOWLETT J., ROTA G. (Eds.). A History of Computing in the Twentieth Century. New York, Academic Press.

[10] DIJKSTRA E. (1975). Correctness Concerns and, Among Other Things, Why They Are Resented. In Proceedings of International Conference on Reliable Software, 546-550. ACM SIGPLAN NOTICES, Vol. 10.

[11] AVITAL S., SHETTLEWORTH S. (1968). Objectives for Mathematics Learning. Toronto, The Ontario Institute for Studies in Education.

[12] POLYA G. (1945), How to Solve It. New Jersey, Princeton University Press.

[13] BACKUS J. (1978). The History of FORTRAN I, II and III. ACM SIGPLAN NOTICES, 13, 8, 165-180.

[14] WEGNER P. (1976). Programming Languages - The First 25 Years. IEEE TRANSACTIONS ON COMPUTERS, C-25, 12, 1207-1225

[15] DIJKSTRA E. (1972). The Humble Programmer. COMM. ACM, 15, 10, 859-866.

[16] SAMMET J. (1978). The Early History of COBOL. ACM SIGPLAN NOTICES 13, 8, 121-161.

[17] JACKS A. (1983). Ten Languages; COBOL. PRACTICAL COMPUTING 6, 4, p.114.

[18] NAUR P. (1960) (Editor). Report on the Algorithmic Language ALGOL '60. COMM. ACM. 3, 229-314.

[19] SAMMET J. (1972). Programming Languages : History and Future. COMM. ACM 15, 7, 601-610.

[20] RADIN G. (1978). The Early History and Characteristics of PL/1. ACM SIGPLAN NOTICES 13, 8, 227-241.

[21] CHRYSLER E. (1980). Computer Programming Productivity. In RULLO T.A. (Ed.). Advances in Computer Programming Management Vol. 1. Philadelphia, Heyden.

[22] JENSEN K., WIRTH N. (1975). Pascal User Manual and Report. New York, Springer Verlag.

[23] HOARE C.A.R., WIRTH N. (1973). An Axiomatic Definition of the Programming Language Pascal. ACTA INFORMATICA 2, 4, 335-355.

[24] HALL J. (1962). (Ed.). Computers in Education. Oxford, Pergamon Press.

[25] KURTZ T. (1978). BASIC. ACM SIGPLAN NOTICES 13, 8, 103-118.

[26] OECD (1971). Report of Seminar on Computer Science in Education. Paris, OECD Publications.

[27] KOFFMAN E.B., MILLER P.L., WARDLE C.E. (1984). Recommended Curriculum for CS1. COMM. ACM 27, 10, 998-1001.

[28] KELLY J. (1984). Why Comal? EDUCATION IRELAND. 1,4, 23-25.

[29] HORTON G. (1985). Structured Programming on the 380-Z. COMPUTER EDUCATION 44, p.22.

[30] BRASWELL J.S. (1984). Advanced Placement in Computer Science. MATHEMATICS TEACHER 77, 5, 372-379.

[31] O'CAOIMH C. (1982). The Computer Studies Option. THE SECONDARY TEACHER 11, 2, 10-11

[32] O'SHEA F.T. (1983). Computers in Schools. EDUCATION IRELAND 1, 1, 20-23.

[33] KELLY J. (1980). Computer Studies 1. Dublin, The Educational Company of Ireland.

[34] O'RINN S. (1983). Is There Life Beyond Computer Studies? EDUCATION IRELAND 1, 1, 25-26.

[35] WIRTH N. (1971). Program Development by Stepwise Refinement. COMM. ACM 14, 4, 221-227

[36] WULF W.A. (1977). Languages and Structured Programs. In YEH R.T. (Ed.). Current trends in Programming Methodology. Englewood Cliffs, New Jersey, Prentice-Hall.

[37] Dijkstra E. (1968). Goto Considered Harmful. COMM. ACM 11, 3, 147-148.

[38] BOHM C., JACOPINI G. (1966). Flow Diagrams, Turing Machines and Languages With Only Two Formation Rules. COMM. ACM 9, 5, 366-371.

[39] DIJKSTRA E. (1975). Guarded Commands, Nondeterminancy and Formal Derivation of Programs. COMM. ACM 18, 8, 453-457.

[40] McCRACKEN D. (1973). Revolution in Programming. DATAMATION Dec.1973, 50-52.

[41] SHEIL B.A. (1981). The Psychological Study of Programming. ACM COMPUTING SURVEYS 13, 1, 101-120.

[42] SCHNEIDERMAN B. (1980). Software Psychology. Cambridge, Winthrop.

[43] MOURSOUND D. (1984). More Harm Than Good? THE COMPUTING TEACHER 12, 4, 3-4.

[44] BRUNER J.S. (1966). Toward a Theory of Instruction. Cambridge, Harvard University Press.

[45] NICHOLLS J.E. (1975). The Structure and Design of Programming Languages. London, Addison Wesley.

[46] RILEY D. (1981). Teaching Problem Solving in an Introductory Computer Science Class. ACM SIGCSE BULLETIN 13, 1, 244-251.

[47] CAMPBELL P.F. (1984). The Effect of a Preliminary Programming and Problem-Solving Course on Performance in a Traditional Programming Course for Computer Science Majors. ACM SIGCSE BULLETIN 16, 1, 56-64.

[48] METZLER R.C. (1984). IF Rules Then Better Structured BASIC. THE COMPUTING TEACHER 12, 4, 12-14.

[49] KELLY J. (1983). Foundations in Computer Studies with COMAL. Dublin, The Educational Company of Ireland.

[50] ATHERTON R. (1982). Structured Programming with COMAL. Chichester, Ellis Horwood.

[51] O'LEARY P., MAXWELL M. (1980). BASIC Computer Programming for Students. Dublin, Folens.

[52] CHRISTENSEN B. (1982). Beginning COMAL. Chichester, Ellis Horwood.