TEACHING COMPUTER CONTROL APPLICATIONS - A PROGRAMMING APPROACH

by

Wai Hing Chung

Ph.D. Thesis

University of Edinburgh

1986

ABSTRACT

The objective of this project is to investigate the task of teaching control applications and the related concepts to secondary school students between the age of 14 and 16. The age range is chosen so that students may leave school with some understanding of the use of computers for automation.

This study is based on a programming approach. A suitable programming language is used by a student to construct programs that control different devices. This approach is appropriate because it helps a student to focus on thinking about the control processes and learn about the functionality of the electronic components commonly used in control devices, without having to worry about the details of electronics. A program represents at least part of the student's understanding of the control concepts used in an application, and his solution may be validated by running the program on a computer.

The idea of learning through programming has been investigated in some depth in the context of mathematics teaching. An important point which has been repeatedly emphasised by the investigators is the suitability of the programming language. To allow a student to be creative and be able to describe his solution or experiment with ideas in a convenient way, the language should provide commands, control and data structures for handling all the kinds of problems likely to arise in a particular domain. With this in mind, a computer language, Concurrent-Logo, has been designed and implemented. It is an extension of Logo, a well developed language for teaching purposes. The novel facilities of Concurrent-Logo

include commands for detecting signals, commands for actuating switches and stepping motors, and a multi-programming capability.

A pilot study has been carried out in a secondary school with two small groups of students. It has three themes:

(1) Development of ideas for a course in control applications.

(2) Evaluation of the language and the course.

(3) Identification of difficulties that students face in learning control applications.

A course consisting of six projects was developed. Each project involved writing programs for a particular control device. The control devices comprised windmill, turtle, doll's house, lift, turtle with optical sensors, and robot arm. These devices make use of a wide range of electronic components and the programming tasks also cover a wide range of control and computing concepts.

From the profile of the students' work some misconceptions that they had and errors that they made are identified. A final test also showed that the students had gained some understanding in control applications. The implementation of Concurrent-Logo was reliable and effective. The students' feedback showed that the course was a success. They enjoyed it and felt that they had benefited from it.

ACKNOWLEDGEMENTS

I am in debt to my supervisors Jim Howe and Peter Ross for their encouragement and supervision throughout the whole project. They, along with Paul Brna, Rosemary Candlin, Bert Hutchings, Ena Inglis, Ken Johnson, Hamish Macleod and Mike Sharples have commented on a draft of this thesis. Andy Russell, Mike Irving and David Wyse helped me design and build the electronics modules and control devices used in this project.

I would like to thank Peter Bates, the Principal Teacher of physics in Firrhill High School, for his willing cooperation. I am also grateful to the students who participated in the study. Economic and Social Research Council supported me financially.

Finally I would like to dedicate this thesis to my fiancee Anissa Ho for waiting patiently for me to complete it.

Declaration

I composed this thesis and the work which it describes was done by myself.

CONTENTS

1. Introduction 1
   1.1 Setting the scene 1
   1.2 Motivation 3
   1.3 The subject 5
   1.4 The needs 11
   1.5 The study 17
   1.6 Layout of the thesis 21

2. Control applications in schools 23
   2.1 Control hardware 24
   2.2 Programming language 31
   2.3 Teaching material 36
   2.4 Teaching methods 37
   2.5 Other uses of control applications in schools 38
   2.6 Conclusion 39

3. Learning through programming 40
   3.1 Programming aids understanding 40
   3.2 Logo 44
   3.3 Other languages 56
   3.4 Learning control applications through programming 71

4. Concurrent-Logo 76
   4.1 Design 76
   4.2 I/O handling 79
   4.3 Control structures 82
   4.4 User defined objects 83
   4.5 Multi-programming 88
   4.6 Event handling 102
   4.7 Related work 106

5. A pilot study 110
   5.1 Aims 110
   5.2 Design 110
   5.3 Participants 112
   5.4 Equipment 115
   5.5 Course 121
   5.6 Evaluation 132

6. Learning control applications through programming 135
   6.1 The students 135
   6.2 An overview 136
   6.3 Windmill 152
   6.4 Turtle I 155
   6.5 Lift 161
   6.6 Doll's house 169
   6.7 Turtle with opto-sensors 175
   6.8 Robot arm 184
   6.9 Discussion 191

7. Students' understanding 197
7.1 Components 198
7.2 Feedback 199
7.3 Robot 200
7.4 Procedures 201
7.5 Parallelism 203
7.6 Conclusion 206

8. Students' response 210
8.1 Course 210
8.2 Projects 213
8.3 Discussion 216

9. Assessing Concurrent-Logo 219
9.1 Programming language design 219
9.2 Concurrent-Logo 228
9.3 Conclusion 239

10. Conclusion 240
10.1 Summary 240
10.2 Criticism 242
10.3 Future research 244

Appendix I : Implementation of Concurrent-Logo 246
Appendix II : Worksheets 262
Appendix III: Questionnaires 295
Appendix IV : Control applications test 308

Reference 319

TABLE OF FIGURES

Figure 1.1 Open loop system 6
Figure 1.2 Closed loop system 6
Figure 1.3 Subjects related to control systems 6
Figure 2.1 Programs for Buggy 27
Figure 2.2 Windmill 30
Figure 3.1 Relationship between building models and writing programs 43
Figure 3.2 Logo and the curricula 48
Figure 3.3 Interactions between programming modes 50
Figure 3.4 Mutually exclusive classes 65
Figure 3.5 Hierarchical classes 66
Figure 4.1 Controlling DC motor 84
Figure 4.2 Turtle 90
Figure 4.3 Doll's house 92
Figure 4.4 Lift 99
Figure 4.5 A multi-programming solution to the lift problem .. 100
Figure 5.1 Fourth year group 114
Figure 5.2 Third year group 115
Figure 5.3 Armdroid 118
Figure 5.4 Stepping motor module 120
Figure 5.5 Windmill project 123
Figure 5.6 Turtle I project 124
Figure 5.7 Lift project 125
Figure 5.8 Doll's house project 126
Figure 5.9 Turtle II project 127
Figure 5.10 Robot arm project 128
Figure 5.11 Fourth year group time table 130
Figure 5.12 Third year group time table 131
Figure 6.1 Turtle tracks 143
Figure 6.2 Turtle paths 145
Figure 6.3 Binary patterns 146
Figure 6.4 Sequence of robot arm positions (I) 149
Figure 6.5 Turtle I connection 155
Figure 6.6 Lift connection 162
Figure 6.7 Doll's house connection 170
Figure 6.8 Turtle II connection 175
Figure 6.9 Robot arm connection 184
Figure 6.10 Sequence of robot arm positions (II) 187
Figure 6.11 Sequence of robot arm positions (III) 189
Figure 7.1 Overall result 207
Figure 7.2 Summary of students' results 209
Figure 8.1 Students' opinion of the course 211
Figure 8.2 Students' opinion of the worksheets 212
Figure 8.3 Students' own evaluation of how much they had benefited 212
Figure 8.4 Students' recommendation of the course 213
Figure 8.5 Students' rating of the devices 214
Figure 8.6 Students' opinoin of amount of time spent on each project 216
Figure 9.1 Facilities used in Concurrent-Logo 229
Figure 9.2 Commands in object and in procedure formats 232
Figure 9.3 Students' preference of formats 233

CHAPTER 1

INTRODUCTION

1.1 Setting the scene

With the growing use of computers in our society, educational computing is becoming increasingly important. Many developed and developing countries have initiated national schemes for introducing computers into secondary and primary schools. To give an example of a national scene, in Britain in 1973 the government sponsored the National Development Programme in Computer-Assisted Learning, costing £2.5 million (Hooper, 1977). In 1980 the Departments of Education and Science of England, Northen Ireland and Wales started funding the Microelectronics Education Programme (MEP). The Programme is 'concerned with microelectronics applications in schools and in non-vocational course for 16-19 year-olds in further education, including GCE O and A level courses, and courses leading to pre-vocational qualifications' (DES, 1981). The Programme has two parts. One is 'the investigation of the most appropriate ways of using the computer as an aid to teaching and learning.' The other is 'the introduction of new topics in the curriculum, either as separate disciplines or as new elements of existing subjects.' The budget for this programme is £8 million. The Scottish Microelectronics Development Programme (SMDP) was set up in Scotland around the same time also with the aim of promoting the use of computers in schools. In 1981 the Department of Industry started a Micros in Schools Scheme. It offered a 50% subsidy for the first microcomputer bought by any secondary school. The scheme was later extended to primary schools and schools for the handicapped. A survey carried out in March 1984 by BBC and MEP showed

that the average number of computers in UK secondary schools was nine and the maximum number in any school was thirty-eight (DES, 1985).

Computers are used in many ways to assist the learning of traditional curricula in schools (Howe & du Boulay, 1979; Feurzeig et al, 1981). Some common uses are running applications programs that compute the results for given problems; running simulation programs that represent an event or system; running drill and practice programs that ask students to type in answers to given questions and then check the results, and running tutorial programs that teach basic concepts. The most common use of computers is for teaching computer studies from general computer awareness to specialised computer science. The UK examination entries for computer studies have increased from 23182 in 1977 to 79009 in 1982 (Weston, 1984).

This thesis is concerned with the teaching of one particular application area of computing, namely control of processes and machinery. Central to the study is the idea that new knowledge and skill may be naturally gained through programming. The idea is in turn inspired by research in Artificial Intelligence.

The contribution of Artificial Intelligence research to education is threefold (Howe, 1978; O'Shea and Self, 1983; Yazdani, 1984). First, the knowledge representation and modelling techniques developed in A.I. open an avenue for developing intelligent tutoring systems that are far more advanced than the normal drill and practice CAI programs (see Sleeman and Brown, 1982). Secondly is the idea that building computer programs is an effective way of testing one's understanding of a complex process, such as solving a mathematical problem, diagnosing a disease or playing chess. In other words,

programming helps a learner to focus attention and gives concrete forms to abstract ideas. With a suitable programming environment a student can engage in purposeful activities. Thirdly, closely associated with the second, is the development of A.I. programming languages suitable for students to explore and experiment with ideas.

The best known computer language that is designed for learning through programming is Logo. The commands for controlling a Turtle are its distinctive features. A Turtle is either a motorised cart with an attached pen or a graphical object on the display screen. The Turtle can be told to move FORWARD <a distance>, BACKWARD <a distance>, RIGHT <an angle> and LEFT <an angle>. These commands are especially suited for learning and exploring geometry (Abelson and diSessa, 1981). A student can easily relate the Turtle's movement with his own. It requires no knowledge of the Cartesian coordinate system. Programming the Turtle to trace out patterns is fun and by doing so a student can naturally extend his existing knowledge to include geometrical concepts such as interior angles, exterior angles, side lengths and symmetry.

Just as Logo makes learning mathematics fun and accessible, this thesis investigates applying the programming approach to the teaching of Computer Control Applications at secondary level.

1.2 Motivation

In industry, there is increasing use of microprocessors and computers in automated manufacturing and control of processes. The teaching of Computer Aided Manufacturing, Robotics or similar subjects in further education has gained much importance and

popularity in recent years. The need to introduce the teaching of control applications at secondary school level is also becoming greater.

The Schools Committee Working Party, of the British Computer Society, wrote (1982) '.... if a school undertakes an overall review of the curriculum during the next few years, the following list of topics should be considered for inclusion as a priority, reflecting as they do the environment and society in which our students will inevitably develop.' One of the topics mentioned is the Automatic Control of Processes.

One of the stated aims of MEP is 'to help schools to prepare children for life in a society in which devices and systems based on microelectronics are commonplace and pervasive' (DES, 1981)

As the demand for trained personnel in control applications becomes higher, industrial companies are also keen to promote interest in the subject so that students will be encouraged and enabled to develop their potential to become creative engineers. In 1980, Fisher Controls Limited took the initiative in collaboration with the Leicester Education Authority (Higgs, 1980). They invited schools in Leicestershire to submit ideas for control projects. The submissions were then assessed. The selected schools received financial and technical support from the company and the education authority to complete the proposed projects.

A 'Buildarobot' competition was sponsored by British Petroleum in 1982 (BP, 1982). The competition was carried out in two phases. Schools were first invited to submit feasibility studies. The best

projects were selected and received £100 towards the cost of making the robot. The competition attracted tremendous interest; over 200 schools in the UK had submitted entries and 21 of them were selected.

Despite all this interest and activity, so far there is little investigation into the teaching and learning of control applications. It would be fair to say that the industrial schemes described above are aimed, perhaps not exclusively, at the more able pupils. To reach a wider population, research into the methods of teaching the subject and systematic course development is essential.

1.3 The subject

Control applications is related to a number of science and engineering subjects. Therefore it may mean different things to different people. Before outlining the problems and the resource requirements of the teaching of control applications it is appropriate to describe briefly the nature and the essentials of the subject matter.

Central to control applications is the idea of a system. In general, control systems are classified into two types: open loop systems (figure 1.1) and closed loop systems (figure 1.2) The difference between them is simply that a closed loop system makes use of feedback information from the control device to make corrective actions and an open loop system does not.

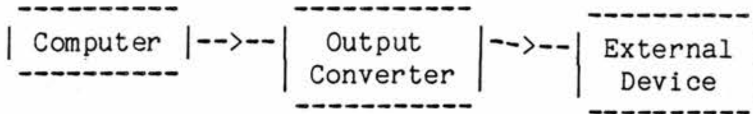


Figure 1.1 Open Loop System

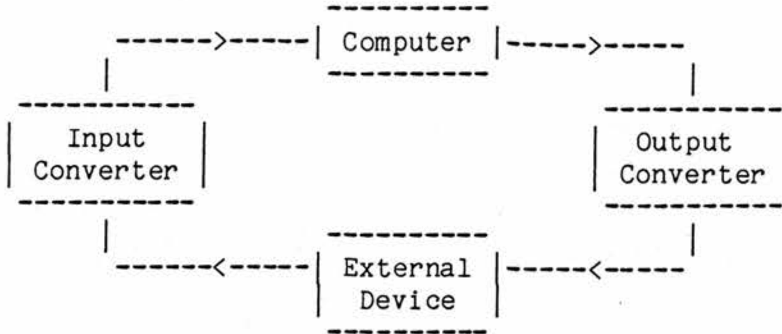


Figure 1.2 Closed Loop System

The study of a control system may be related to many different subjects. Figure 1.3 shows the relationship.

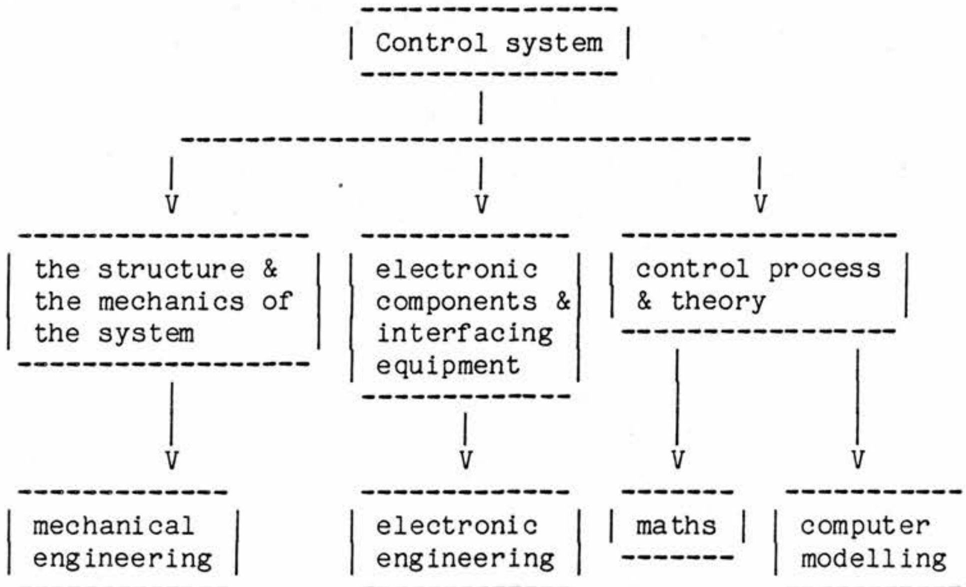


Figure 1.3 Subjects related to control systems

At the introductory level, it is inappropriate to clutter a course in control applications with detailed mathematics, electronics and computing. Instead the course should help a student to

(1) understand the basic control concepts

(2) understand the computing concepts which are particularly useful in control applications

(3) understand the control algorithms used in different control systems

(4) recognise the common electronic components that are used in control systems.

1.3.1 Control concepts

Input, output, state, feedback and sampling rate

A control system usually has a set of variables and the complexity of the system increases as the number of variables increases. The state of the system is determined by the values of these variables. The controller, i.e. the computer, reads input from various sensors to find out the current state of the system. The controller then compares the current state with the desired state. If there is a difference, the controller sends output to activate, or deactivate, certain components to bring the system into the desired state. This use of information produced at one stage of processing as input at another is called feedback. A control system that makes use of feedback information to make decision and take corrective actions is called a closed loop system (see fig 1.2). In general, closed loop systems are divided into two types (Marshall, 1978):

(1) regulator - a system which has to maintain an output equal to a desired value despite outside changes and disturbances, for

example, a central heating system.

(2) servomechanism - a system which has to produce an output position equal to some reference input position, for example, a lift.

The stability of a regulator and the accuracy of a servomechanism are dependent on the sampling rate, i.e. the number of times that input is read and processed in a fixed time unit. In some applications, failure to compute a result within a specified time may be just as bad as computing a wrong result.

1.3.2 Computing concepts

One characteristic of control applications is that the computer has to respond to ongoing processes. As described previously, the role of the computer is to read input from sensors to find out the current state of the system; and send output to certain components to bring the system to the desired state. The computing concepts which are particularly relevant are input, output, conditional execution and parallel processing.

In a closed loop system, the output values depend on the input values. Therefore, the computer must examine the input values and decide on the actions that are to be taken. The idea of testing whether certain conditions are true and then taking the appropriate course of action is called conditional execution. Programming languages provide conditional statements for this purpose.

Parallel processing is relevant in the control applications context because it is common for a control system to have several components that require attention at the same time. It is

appropriate to use parallel processing to deal with inherently parallel processes. However, parallel processing is a vast subject and much research is still being done. The aim, therefore, is not to teach it thoroughly but to give students the access to the use of it. Through the practical experience they may appreciate the power of and some of the problems with parallel processing.

1.3.3 Control algorithms

Besides teaching the relevant control and computing concepts, it is important to help the students to appreciate how these concepts are applied in some particular systems. Control algorithms is the study of how the behaviour of different systems is achieved.

The following example considers using different algorithms to control a simple lift system which has three levels. A basic function of the lift would be to move the lift cage up and down cyclically between the bottom and top levels. Assuming the lift cage is at the bottom, the algorithm would be:

```
Forever (
    (1)  start the lift cage moving upward;
    (2)  wait for a specified amount of time, i.e. the time
         required to move the cage from the bottom to the top.
    (3)  start the lift cage moving downward;
    (4)  wait for a specified amount of time, i.e. the time
         required to move the cage from the top to the bottom.
)
```

The algorithm is inflexible because it depends on the absolute time required to move the lift between two levels. The algorithm could be made slightly more general by using feedback information from sensors at the bottom and top of the lift:

```
Forever (
    (1) start the lift cage moving upward;
    (2) wait until sensor at top level is on;
    (3) start the lift cage moving downward;
    (4) wait until sensor at bottom level is on;
)
```

A simple and realistic extension to the function of the lift would be to add three button switches to the system so that

(1) whenever switch 1 is pressed, move the cage to the bottom level,

(2) whenever switch 2 is pressed, move the cage to the middle level,

(3) whenever switch 3 is pressed, move the cage to the top level.

Although the functional extension is trivial, the control process that would achieve this result is already quite complex. The computer needs to monitor the states of the button switches continuously and activate the movement of the lift cage accordingly. Moving the cage to the second level would require information about its current position to decide whether to move it up or down. By adding further scheduling requirements the control process could be very complex indeed. As the example shows, the lift may be programmed to do apparently the same thing (move up and down), but the underlying control processes differ greatly.

Different systems usually share certain characteristics, such as the use of feedback information or the use of similar electronic components. However, each system also has its intrinsic characteristics. For example, a lift is very different from a robot

arm. A course in control applications should cover a wide range of applications and help the students to understand the different algorithms used.

1.3.4 Electronic components

It has been mentioned that a course in control applications should not be cluttered with detailed electronics. However, the students should be given the opportunity to program simplified or scaled-down versions of real-life control systems. This will the students a sense of realism. Furthermore, they will gain familiarity with some of the sensors and actuators that are used in control systems.

1.4 The needs

Secondary school students, instead of developing a fear, should be encouraged to develop an appreciation and understanding of control applications.

At present, control applications is usually taught to 11-13 year old students as part of a technology awareness course, and to older students as part of an O/A level course in electronics, computer studies or related subjects. The problem with the way that control applications is being taught is that it is introduced at either too low or too high a level. At the low level end the emphasis is on understanding electronics and machine oriented programming. At the high level end the emphasis is on demonstrating what a control device can do and neglects the aspect of how it does it. As mentioned earlier the teaching should be control process oriented. The students should be led to focus on thinking about how the functionality of a

control system is achieved. They should be led to ask questions like: what information is required? What events should be monitored? What should be the response? How should we achieve it?

To develop a suitable control applications course, there are three equally important areas of concern:

(1) hardware; the physical devices and all the necessary components for connecting the devices to a microprocessor and a computer.

(2) programming language; a language that is understandable by computers and allows a programmer to express control algorithms conveniently.

(3) courseware; a collection of books and worksheets for use by students and teachers.

With the growing interest in control applications an increasing variety of control hardware is becoming available. These include specially designed control devices such as Buggy[1], Armdroid[2] and general purpose interfacing hardware (Andrew and Whittome 1981).

1.4.1 Programming language

The main restriction is that there is no suitable programming language. Students are forced either to learn electronics and low level programming or to be content with observing and running demonstration programs.

The suitability of a programming language may be judged on two grounds: suitability for the users, who in our context would be

---

[1] Buggy is manufactured by Economatics Ltd, Sheffield.
[2] Armdroid is manufactured by Colne Robotics, London.

secondary school students and teachers, and suitability for the application. There are differences between the requirements of a programming language for learning and one that is for use by professionals in industry. An analogy will make this point clear: 'We might expect to find that racing drivers are impatient with slow, family saloon cars, though the latter are much better than racing-cars for the learner drivers.' (du Boulay et al, 1981). Ross and Howe (1984), recalling some of the principles and decisions that contributed to the design of the Edinburgh version of Logo, point out that it 'was initially aimed at children and the classroom, and so the normal priorities in the design of the language had to be reshuffled somewhat.' Designing a programming language for learning might be called 'cognitive engineering' (Lawler, 1984), for its purpose is to shape children's minds. In general a programming language for novices should be interactive, extensible, visible and simple.

An **interactive language** allows a student to test or try out his ideas easily, and provids immediate feedback. This conflicts sharply with the crucial requirement of 'security' in system development languages. 'The security of a language is a measure of the extent to which programming errors can be detected automatically by the compiler or language run-time support system....... In general, the design of a language should be such that as many errors as possible are detected at compile-time rather than at run-time.' (Young, 1982). The means by which errors may be detected at compile-time forces the programmer to provide extra information for the compiler. For example all the variables used in a program have to be declared and their types have to be specified. To use a compiled language, one

also has to learn to use an operating system. All these imposed demands can be discouraging, particularly for novice programmers for whom the benefits of interactiveness far outweigh the benefits of security.

An **extensible** language is one that allows a programmer to create new procedures which can be used in exactly the same way as system provided procedures. This facility encourages a good approach to problem solving. A student is able to build up a set of higher level procedures, and to combine them easily to make more complicated programs. Another advantage is that a teacher can introduce a problem at the right level of abstraction, hiding all the unnecessary details by providing higher level procedures. A student can run them, become familiar with them and later adopt them to solve complex problems.

A **visible** language makes it easy for a student to understand and to follow what a computer is doing when executing a program. The former aspect of visibility is concerned with providing a simplified model - a notional machine - of how the computer works at the level of the operations of the programming language. The latter aspect of visibility is concerned with providing commands in a language that provide immediate feedback, such as sound, visual display or movement of a device.

A **simple** language is one that is easy to learn and to use. Dijkstra (1972) wrote: 'the development of "richer" or "more powerful" programming languages was a mistake in the sense that these baroque monstrosities, these conglomerations of idiosyncrasies, are really unmanageable, both mechanically and mentally'. However, a

simple language does not mean a restricted and unexpressive language.

There are two aspects of simplicity (du Boulay et al, 1981): syntactic and logical. Syntactic simplicity means that the rules of the language are simple and consistent. There should be very few exceptional cases and no ambiguities, thus making the language easy to learn. Logical simplicity means that the language allows a student to describe the structure and mechanism of his solutions in a convenient way, helping him to focus attention on solving problems rather than on the peculiarities of the language. This implies that the language should provide suitable primitives, and also control and data structures for handling all the kinds of problem likely to arise in a domain which he is investigating. A programming language that has all these characteristics would make it easy for a student to do interesting things without having to overcome many initial hurdles and it would enable him to experiment easily with different ideas in control applications.

In comparison with languages developed for other applications, a control language also has three distinct features:

(1) I/O handling

(2) Event handling

(3) Multi-programming

In control applications, it is necessary for a computer to communicate with external devices. Facilities must be provided for programming devices conveniently. The computer must also detect the occurrences of some events and respond to them. A control system usually consists of several active components that require attention from the computer. They are naturally described as a set of

concurrent processes, and a control language should provide some facility for multi-programming.

In view of the general and specific requirements, there is currently no suitable computer language for learning control applications. Although BASIC, the most commonly used programming language in schools, is interactive, it is neither extensible nor simple. Dijkstra (1982) wrote, 'It is practically impossible to teach good programming to students that have had a prior exposure to BASIC: as potential programmers they are mentally mutilated beyond hope of regeneration' (See Section 2.2.). Machine oriented assembler languages are also far from satisfactory.

1.4.2 Course

The lack of a suitable control language also limits the range of control projects that students can do and hence hinders the development of courseware. The six principle design criteria laid down by MEP for the course 'Microelectronics For All' are relevant here:

(1) Must be acceptable, stimulating and meaningful to pupils of all abilities.

(2) Must not require specialist technical or scientific knowledge on the part of the teachers.

(3) Must be practically based with all pupils having the opportunity to investigate systems which they can understand, describe and modify. Ideal pupil group size working together 2 and not to exceed 3.

(4) The course must provide an introduction to the subject and

a foundation for those who choose to study the subject beyond this level.

(5) Must be based on a closely defined core practical programme tailored to lower abilities which is extendable to the degree required for brighter pupils.

(6) The core programme to be supported by a complementary teaching programme designed to relate the practical classroom experience to real applications in industry, commerce and the home and to develop an awareness of the implications of current and future development.

Beside the immediate need for course development, another important area of research is to build up a body of knowledge of how students learn control applications. In order to improve the teaching of control applications, it is necessary to know what kind of misconceptions students hold, what common mistakes they make and so on.

1.5 The study

The study had four related themes:

(1) Design of a programming language suitable for learning control applications.

(2) Development of ideas for a course in control applications.

(3) Evaluation of the language and the course through an experimental study.

(4) Identification of difficulties that students face in learning control applications.

The approach to the design of the computer language was by first choosing a language which has the structures suitable for the potential users, i.e. secondary school students and teachers, and then incorporating into the language primitives suitable for the application. Four languages were considered: Logo, Prolog, Smalltalk and Forth (see chapter 3).

Logo (Abelson, 1982) was chosen as the base language because it is interactive, extensible and has a simple and consistent syntax which make it a good language for learning; it is procedural which makes it a suitable language for describing processes. The extensions to the language include commands for detecting signals, commands for actuating switches and stepping motors, and also abstract data types, multi-programming and guards (see chapter 4). The extended language is called Concurrent-Logo (Chung, 1984).

A course consisting of six projects was developed. Each project involved writing programs for a particular control device. The control devices were: windmill, turtle, doll's house, lift, turtle with optical sensors and robot arm. The course was designed to

(1) give the students practical experience in using the basic control concepts: state, feedback and pulsing.
(2) give the students practical experience in using programming constructs such as: procedures, conditionals and parallel processing.
(3) help the students understand how the devices work
(4) familarise the students with components which are commonly used in control devices: DC motor, stepping motor, button switch, reed switch, reflective-opto switch and micro switch.

Howe and Delamont (1974) distinguish two stages in evaluating educational innovations. The first, termed monitoring, gathers general impressions on the innovation with a view to improving it. The second, termed non-reactive, collects data in a controlled fashion using rigorous techniques. The development of Concurrent-Logo and the course were at the pioneering stages. Therefore a small scale formative evaluation study was most appropriate. The results from a formative study should help to identify some common difficulties that the students faced, to refine the course and the language, and to form more specific questions for future research. Because the author had to be both the teacher and the investigator, the number of students in each class needed to be reasonably small to allow him to make observations without being distracted too much by the teaching.

The course was taught to two groups of students from Firrhill High School, a secondary school in Scotland. One group of five students was from the fourth year (approximately fifteen years of age) and the other group of seven students was from the third year (approximately fourteen years of age). Over two school terms, the fourth year group had seventeen sessions and the third year group had fourteen sessions. Each session lasted 75 minutes. The fourth year group had fewer sessions because they had to spend more time preparing for their examinations.

The central classroom activity was the students programming the computers. Worksheets with explanations and suggestions for programming projects were provided. A structured teaching strategy was adopted because it would give the students the same starting

point and help them to acquire a common vocabulary and a common set of concepts very quickly. Furthermore, designing programming tasks for a control device is not trivial. It requires some appreciation of what the device can do and the different ways that it may be controlled in the first place. For example, it may be obvious to a student that a lift may be programmed to move up and down but he may not be aware of the varieties of control strategy that may be used. Therefore there need to be well planned suggestions that would lead the students from one programming task to the next. Through programming they might acquire an appreciation and understanding of the different embedded control concepts.

Throughout the course, four questionnaires were designed and used. During each session the students' work was recorded: everything that they typed was recorded on disk in addition to the author's own observations on paper. At the end of the course the students were also given a test. From the profile of their work some misconceptions that the students had and errors that they made were identified. The test results showed that the students had gained some understanding in control applications. The survey also showed that the course was a success. The students enjoyed it and felt that they had benefited from it.

The implementation of Concurrent-Logo proved reliable and effective. It helped the students to focus attention on solving control problems and minimised distractions due to peculiarities of the language or details of computer hardware. Some extension to the language is suggested, so that the language may be used for a wider range of projects.

1.6 Layout of the thesis

The rest of this thesis has nine chapters and two appendices. Chapter 2 reviews the state of the art of the teaching of control applications in schools. The review considers the resources that are available and the ways that the subject is being taught.

Chapter 3 outlines the rationale behind the methodology of learning through programming and reviews four interactive computer languages, namely: Logo, Prolog, Smalltalk and Forth. Their suitability as languages for learning control application is considered; Logo is chosen as the most suited for extension.

Chapter 4 explains the overall design philosophy of an extended Logo - Concurrent-Logo - which is specifically designed for teaching and learning control applications. The novel features of Concurrent-Logo are described with examples of their use.

Chapter 5 gives the design details of the previously mentioned pilot study carried out in Firrhill High School.

Chapter 6 describes and analyses the work done by four of the students who took part in the pilot study. Their work is fairly representative of the work done by all of the students. The description concentrates on

(1) the variation of the students' work

(2) the difficulties that the students faced

(3) how the students solved the problems.

The chapter ends with a summary of the benefits and limitations of learning control applications through programming.

Chapter 7 assesses the students' understanding of control and related concepts. At the end of the pilot study a test was designed and given. The students' answers are categorised into different levels of understanding where possible.

Chapter 8 describes the students' opinion concerning the course. The information is obtained from questionnaires filled in by the students at the end of the pilot study.

Chapter 9 evaluates Concurrent-Logo. Features of the language that are likely to cause programming errors are identified. Further extensions for the language are also discussed.

Chapter 10 summarises the results and the limitations of the study. This final chapter ends with several suggestions for future research.

Appendix I is a brief description of the implementation of Concurrent-Logo. The formal syntax of Concurrent-Logo is also described in Backus-Naur form.

Appendices II and III are worksheets and questionnaires respectively, which were produced for and used in the study.

Appendix IV is a control applications test. Its design is based on the experience gained from the study.

Throughout the thesis, in-line comments for program listings are preceded by the symbol '@'.

CHAPTER 2

CONTROL APPLICATIONS IN SCHOOLS

Papert observed that the teaching of control applications and the use of robots in schools are much more common in Britain than in the United States (Ginn, 1984). He suggested two reasons:

(1) it is Britain's tradition to use very concrete objects for introducing young children to abstract thinking.

(2) in Britain manufacturers made robots earlier than they made computers with good graphics.

A third, perhaps a more important, reason is that in Britain there is a recognition that control applications is an important subject. This is manifested in several ways. MEP, a government sponsored project, has set up a 'Control Technology Domain' to provide in-service training for teachers (Bevis, 1984). Industry is promoting interest in the subject by organising competitions for secondary schools. O and A level courses in computer studies and electronics are evolving to include more and more control aspects. However, this does not mean that all is well.

This chapter reviews the current state of the art of the teaching of control applications in Britain. This review begins by looking at the resources that are available. The first section is concerned with hardware. The second section is concerned with programming languages. The third section is concerned with teaching materials. Then, in the fourth section, the methods of teaching control applications in schools are considered. Section 5 describes some uses of control applications to assist the teaching of other science subjects.

The conclusion is that control hardware development has made considerable advancement whereas there is relatively little effort put into the development of a computer language for learning control applications. As a result the development of teaching material is hindered and the classroom activities restricted.

2.1 Control hardware

In Britain, a lot of effort has been put into developing control hardware for educational purposes. The hardware falls into two categories: special purpose control devices and general purpose modules. The main types of control devices are mobile robots that move around on wheels, and robot arms. These devices are either ready built or in kit form. With each device specially designed interfacing and power supply boxes are provided by the manufacturers so that connecting a computer with the device would be straightforward. Instead of providing specially designed control devices, another way is to provide general purpose modules that facilitate the construction of control devices and the task of connecting the devices to a computer.

2.1.1 Turtle and derivatives

The Turtle is a computer controlled, motorised cart with an attached pen. It is called a turtle because of its shape. The Turtle was invented at MIT, as a tool to introduce young students to the ideas of problem solving and mathematics (Papert, 1971a). The Turtle responds to commands, either to move FORWARD or BACKWARD in the direction it is currently facing, or to rotate LEFT or RIGHT on the spot. These commands are a subset of the Logo programming language.

They are body centred and a student can easily relate the turtle's movements with his own. It serves as an 'object-to-think-with'. Listing 2.1 shows a sequence of commands for drawing a square.

```
FORWARD 100
RIGHT   90
FORWARD 100
RIGHT   90
FORWARD 100
RIGHT   90
FORWARD 100
RIGHT   90
```

Listing 2.1 SQUARE program

Besides drawing, a student might program it to follow a path or to knock a pile of bricks over.

The idea of using the Turtle for teaching control applications was first suggested by Papert (1971b). He suggests attaching sensors to the turtle to provide feedback information for the computer. A turtle with optical sensors[3] could be programmed to follow a track; a turtle with touch sensors could be programmed to walk around obstacles or to find its way out of a maze. Turtles are now produced by several companies. The following describes two variants of turtles.

Buggy

Buggy is a turtle-like robot. It was developed by MEP for the BBC Computer Literacy Project as a complement to the series 'Making the most of the Micro'. It is designed especially to be used with a

[3] The version of the Turtle manufactured by Terrapin Inc., U.S.A., has touch sensors mounted on it.

BBC model B computer. A Buggy can be connected directly to the user and analogue ports of a BBC model B computer.

The sensors mounted on a Buggy include: microswitches for detecting collisions and an opto-reflective sensor for sensing whether the Buggy is on a black or white surface. There are options for a pen-up, pen-down mechanism and a gripper.

Buggy is available in kit form so that students can learn from assembling it.

Thirteen programs are supplied with the kit. They are designed to demonstrate the basic ideas of programmable control applications. Figure 2.1 gives a summary of these programs. No other device is supplied with so many programs.

Big Trak

Big Trak[4] is a toy cart based on the idea of a Turtle. Its advantage is that it is completely self-contained and operated by batteries. All the electronics and programs for driving the cart are built into the device, so there is no need to connect it to a computer. It is a simple device that can be used in any classroom.

---

[4] Big Trak is manufactured by Milton Bradley Electronics, West Germany.

Program name	Description
TEST	checks that BUGGY is set up correctly and that all sensors are working.
SWITCH	shows the essential commands in BASIC which drive the Buggy.
MEMORY SWITCH	allows the user to drive the Buggy by single key-presses; the computer records the key-presses and will replay them in sequence or in reverse order.
RECORDER	as the user drives Buggy interactively an on-screen map of the path is drawn graphically.
SNAIL	it is like RECORDER, but the sequence of instructions for Buggy has to be input first. As Buggy follows the instructions a trail is drawn on the screen to show the progress.
ROUTEPLANNER	the user first creates a route on the graphics screen, the information on the screen is then interpreted by the program and Buggy then enacts the designed route.
BAR-CODE ROUTE PLANNER	it is like ROUTE PLANNER, but the route information is read by Buggy from Bar-coded cards and then the route enacted.
EXPLORE FOR OBJECT	Buggy seeks out an object, then crawls around it to find out how large it is. The shape of the object is then drawn on the screen.
EXPLORE FOR WALL	Buggy can be placed anywhere in a bordered area. It first crawls around the border and then draws a correctly scaled map of the area and its current position in the area.
SUNSEEKER	Buggy tries to seek out a light sources and get around any obstacles that are in the way.
MAN vs BUGGY	The task is as for SUNSEEKER but Buggy is driven by the user using the same information as would otherwise be available to the SUNSEEKER program.
LINE FOLLOWER	Buggy follows a black line on a white surface or vice versa.
TIN PAN ALLEY	Buggy reads musical information (score) from special bar-codes and the score is then displayed graphically.

Figure 2.1 Programs for Buggy

A student programs the cart by entering turtle-like commands from the key-pad on the top of the cart. It is cheap and convenient. However, like the turtle it can not be used to teach many control concepts. It has the further disadvantage of being inaccurate.

2.1.2 Robot arm

With the fall in hardware prices, small robot arms with reasonable accuracy are also becoming cheaper. This means that every school should be able to afford one in the future.

Examples of robot arms that are developed for educational use are Armdroid, Atlas[5], Ogre 1[6] and MA 2000[7]. They are small and friendly devices for students to work with. Many basic ideas in robotics could be taught using such devices, if appropriate control software and programming languages were available.

A robot arm is usually provided with a teaching program so that a student can operate the arm using single key-presses, record a sequence of actions and replay it. This technique of programming a robot is known as 'teaching by showing' (Motiwalla, 1982). However, manufacturers seldom provide any other software.

2.1.3 Hardware modules

The Advisory Unit For Computer Based Education (Andrews P.J. and Whittome L.J., 1981) has developed general purpose hardware that makes it easy for students and teachers to construct their own model

[5] Atlas is manufactured by L.J.Electronics Ltd, Norwich.
[6] Ogre 1 is manufactured by L.W.Staines & Co., London.
[7] MA 2000 is manufactured by TecQuipment International Ltd, Nottingham.

control devices.

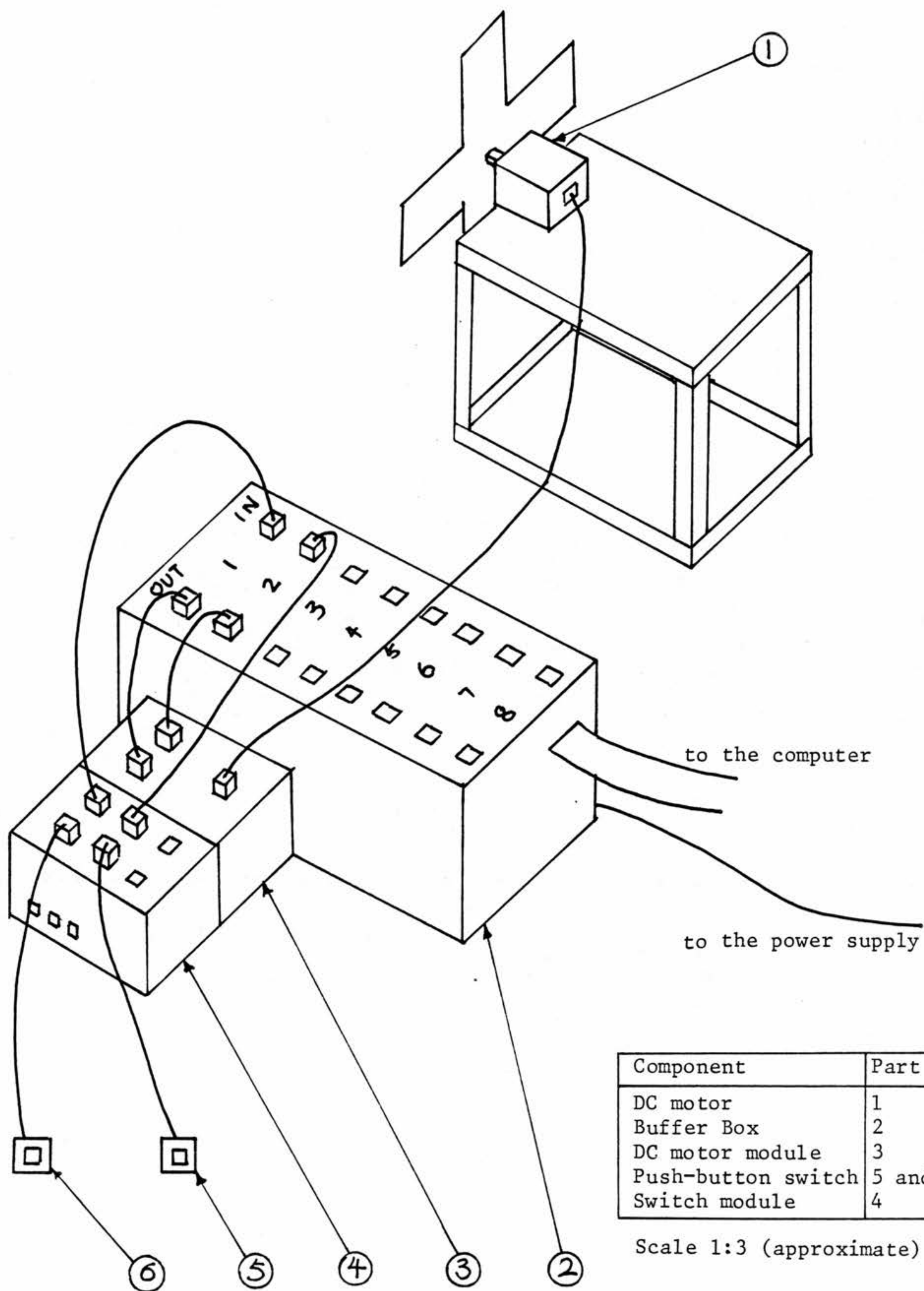
The hardware basically consists of:

(1) a Buffer Box which plugs into the parallel port of a computer. The Buffer Box serves three purposes. First, it has protective circuitry so that any misconnections would not damage the computer. Second, the individual input and output lines of the computer are linked to convenient sockets on the top of the Buffer Box. Third, it is the power supply for other modules.

(2) a collection of modules which plug into the Buffer Box. Each module provides a specific facility. Already over ten modules have been developed. They include modules for push-button switches, light activated switches, bleeper, joystick, analogue/digital converter and DC-motor.

Example: a windmill

Figure 2.2 shows the construction of a simple computer controlled windmill using the DC-motor module, push-button module and Buffer Box.



Component	Part No.
DC motor	1
Buffer Box	2
DC motor module	3
Push-button switch	5 and 6
Switch module	4

Scale 1:3 (approximate)

Figure 2.2 Windmill

The windmill may be built out of any modelling kit, for example Meccano, Lego or Fishchertechnik.

The Buffer Box is first connected to the computer, so that the input and output sockets respectively correspond to the input and output lines of the computer. The motor on the windmill is connected to the DC-motor module. Two output lines are used to drive the motor.

The push-button module has two push-button switches connected to it and the output from them is connected to input sockets of the Buffer Box. These switches are used to start-stop the device.

The computer may be programmed to sense inputs from the button-switches and start or stop the motor by altering the output signals to the motor module.

The module approach provides a convenient and economical way of designing and constructing a wide range of control devices for use in schools. A variety of hardware module systems are now commercially available (for example Beasty[8] and Fischertechnik Robot Kit[9]).

2.2 Programming language

2.2.1 BASIC

BASIC is the most widely used language in secondary schools. It is popular because it is interactive, making it seem easy to use, and it is widely available; most, if not all, inexpensive micro-computers

[8] Beasty is manufactured by Micro-Robotics, Cambridge.

[9] Fischertechnik Robot Kit is manufactured by Fischertechnik, Wimborne.

are provided with a version of BASIC.

Although the original Dartmouth Basic does not have commands for controlling devices, most dialects of BASIC available on micro-computers have the commands PEEK and POKE. 'PEEKing' at the contents of a machine-specific address reads signals from an external device. 'POKEing' a number into the same (or another) machine-specific address sends signals to a device.

In his paper advocating the use of microcomputers in schools, Sparkes (1982) gave as an example a BASIC program (listing 2.2) for controlling traffic lights. The program is written for the PET computer, which is a 8 bit machine with Data Direction Register address 59459 and Parallel Port address 59471. It is assumed that a red, amber and green light are connected to lines 0, 1 and 2 of the Parallel Port respectively. The program simply switches on and off the lights periodically.

```
100 POKE 59459, 7               @lines 0, 1 and 2 as outputs
110 POKE 59471, 1               @switch red on,
                                 amber and green off
120 FOR T = 1 TO 10000: NEXT T  @delay 10 second
130 POKE 59471,3                @switch red and amber on
140 FOR T = 1 TO 2000 : NEXT T  @delay 2 seconds
150 POKE 59471,4                @switch green on,
                                 red and amber off
160 FOR T = 1 TO 10000: NEXT T  @delay 10 seconds
170 POKE 59471,2                @switch amber on,
                                 red and green off
180 FOR T = 1 TO 2000 : NEXT T  @delay for 2 seconds
190 GOTO 110                    @repeat the sequence.
```

Listing 2.2 Traffic Light Program (BASIC)

The program is obscure: it is machine oriented rather than problem oriented. The low level PEEK and POKE commands are awkward to use and would make little sense to pupils who knew no binary

arithmetic.

The Advisory Unit For Computer Based Education has produced a version of Control BASIC (Wood, 1981). It provides commands for setting the individual lines of a parallel port to 'high' or 'low' and for detecting the states of the individual lines. Listing 2.3 is a traffic light program written in Control BASIC.

```
10  SET(0) (1) (2) LOW          @switch off all the lights
20  SET(0) HIGH                 @red on
30  GOSUB 200                   @long delay
40  SET(1) HIGH                 @red and amber on
50  GOSUB 300                   @short delay
60  SET(0) (1) LOW              @red and amber off
70  SET(2) HIGH                 @green on
80  GOSUB 200                   @delay
90  SET(2) LOW                  @green off
100 SET(1) HIGH                 @amber on
110 GOSUB 300                   @short delay
120 SET(1) LOW                  @amber off
130 GOTO 20                     @repeat the sequence
140
150
200 FOR I=1 TO 1000 : NEXT I    @long delay loop
210 RETURN
220
230
300 FOR I=1 TO 200  : NEXT I    @short delay loop
310 RETURN
```

Listing 2.3 Traffic Light Program (Control BASIC)

Although the extended version is an improvement over the standard version, the control commands still bear little direct relationship to control problems. Furthermore, BASIC is a limited educational language due to its fundamental design. The criticisms of BASIC are

(1) it lacks control structures and procedure mechanisms

(2) it is not extensible

(3) it is syntactically ambiguous.

The lack of control structures and procedure mechanisms leads to excessive use of GOTO statements. There is no built-in mechanism to prevent the bizarre use of GOTO statements, such as jumping out of, or even into, FOR loops or subroutines. As a result, a program written in BASIC does not reflect a neat hierarchical structure, and bad programming habits will be reinforced if a programmer is inexperienced and undisciplined.

BASIC is not extensible. User defined programs are distinctly different from the primitive commands and only one program may be in the main memory at any one time. It is not possible to build up a collection of ´building blocks´ for solving complex problems.

du Boulay et al (1981) give as an example of ambiguous syntax in BASIC its use of the ´=´ symbol for multiple purposes, including assigning values to variables and testing for equality.

The first criticism has been overcome partly by the new implementations of the so-called structured BASIC, e.g. BBC BASIC (Coll, 1982) and COMAL (Atherton, 1982). Structured BASIC has been extended to provide the IF...THEN...ELSE and the REPEAT....UNTIL constructs. The procedurisation mechanism is much improved: it allows parameter passing, variables to be local to a procedure, and recursive procedure calls.

Listing 2.4 shows a BBC BASIC program (taken from Bostock (1983)) for making a Buggy move forward until it collides with another object. The program uses the extended REPEAT UNTIL control structure and procedure facilities.

```
10  @Move Buggy forward until collision
20
30  ?&FE62=31
40  port=&FE60
50  wait=10
60  :
70  REPEAT
80  ?port=2 :PROCdelay
90  ?port=0 :PROCdelay
100 UNTIL ?port=128 OR ?port=64
110 :
120 END
130 :
140 DEFPROCdelay
150 FOR delay=1 TO wait: NEXT delay
160 ENDPROC
```

Listing 2.4 A program for Buggy (BBC BASIC)

The program is still obscure. Although it does not use the PEEK and POKE commands it still relies on examining and assigning values to particular machine addresses. The development of structured BASIC has made no real advance in making BASIC a suitable language for teaching control applications.

2.2.2. Assembler language

An alternative to BASIC is assembler language. Sometimes it is used because speed is essential to an application and BASIC is too slow (for example see Stevenson, 1980). Another reason given by Pike (1982) is that 'if any degree of realism with industrial control work was to be achieved then the use of a high level language was thoroughly inappropriate.'

The first reason is justifiable at present because of the limitation of hardware speed. However, this will not be true in the future as the hardware technology is advancing so rapidly. The second reason is absurd. It is like advocating that computing science

students should be taught Fortran and Cobol, instead of other better structured languages, because they are most widely used in industry and commerce. Surely the aims are to teach general principles and enforce good programming habits. Furthermore, even in industry the aim is to move away from low level languages to higher level, more descriptive languages. Concurrent-Pascal (Brinch Hansen, 1975), Modula (Wirth, 1977) and Ada (Goos and Hartmanis, 1983) are all high level programming languages developed for real time applications (of which control applications is a part). The problem is that these languages are sophisticated compiled languages developed for professional programmers.

2.3 Teaching material

No book has been written specifically for teaching and learning control applications at secondary school level. Books for post-A level studies and the hobbyist exist (for example see Johnson et al, 1984; Foster, 1982). They assume that readers have some knowledge of assembler programming and are written for particular microprocessors or microcomputers.

O and A level electronics text books exist and they usually contain very small sections on control applications (for example see Bevis and Trotter, 1981). However, the topic is introduced at a low level. Prior knowledge of logic gates and circuitry is necessary.

A useful source of information for teachers is 'Microelectronics System News', a quarterly journal published by IEE and supported by MEP and the Department of Trade and Industry. The aim of the journal is to keep teachers in touch with electronics and computing in school

education. Articles are written by teachers to share their ideas and to inform others of current development in their schools.

2.4 Teaching methods

The way that control applications is taught depends largely on the expertise, enthusiasm and initiative of individual teachers and groups of students (Bevis, 1984). Three approaches may be identified: demonstration, project and module approaches.

The purpose of the demonstration approach is simply to show the students that the computer can be used to control external devices. A teacher provides examples of working control systems, the students observe the demonstrations and participate in discussions. Student participation may be increased by allowing students to run the application programs. The advantage of this approach is that it does not require much resources, in terms of hardware, time and manpower. Buggy and the set of supplied programs would meet the requirements. However, the approach is limited. Students may acquire some appreciation that computers can be used for control purposes, but because of the lack of involvement it is unlikely that they will acquire any understanding of the subject. They really have no means of finding out for themselves how the system works.

At the other extreme is the project approach. A teacher and a group of students work as a team on a project over a long period of time, possibly several months. The team specifies the full functionality of a control device, designing and constructing the hardware, and writing the control programs. A wide range of projects has been attempted by different teacher-and-student teams. The end

results are very impressive. Example of these systems are: a computer controlled railway (Avis and Else, 1981), an energy conservation system (Howard and Hooton, 1981), a church bell ringing system (Stevenson, 1980), and a home seeking robot (Thompson et al, 1984).

Unfortunately, the project approach is not suitable for normal classroom practice; it requires a high teacher/student ratio, and usually only the able students are selected. To carry out a control application project from beginning to end is intellectually demanding. It requires knowledge and skill in design, electronics, computing and long term planning.

The module approach is structured. A series of activities and worksheets is designed by the teachers and the students follow guide lines (for example see Simmond (1982)). This approach would be restrictive if applied rigidly. However, it provides most scope for students of different abilities if the modules are designed such that each individual student is able to try out ideas up to his own level of competence, at his own pace. A further advantage is that it does not require a high teacher/student ratio.

The difficulty is to design a set of interesting activities and to provide suitable facilities for carrying them out. At the moment, the main obstacle that hinders this approach is the lack of a suitable computer language.

2.5 Other uses of control applications in schools

Besides teaching control applications as a subject it has also been used to assist the teaching of physics. The main idea is to use

the computer to automate certain experiments. One advantage is that it would free students from the tedious task of collecting data. Another is that the result can be displayed on the VDU in different forms, such as a bar chart or graph.

GEIGER (Grant, 1980) is a control program written by a group of physics teachers for assisting the teaching of the half-life of radioactive substances. Some special purpose-built hardware is required to connect a computer to a Geiger-Muller tube and scaler. The program periodically reads and records the radio-activity count. The decay curve can be plotted on the screen as the data are collected so that any apparent anomalies can be discussed as they appear.

Wilson (1984) and Blackburn (1980) also provide examples of programs that automate experiments for studying acceleration due to gravity and the effects of heating and cooling respectively.

2.6 Conclusion

The teaching of control applications in secondary schools is still at its infant stage. Some advances have been made in hardware development. A range of control devices is available. The major contribution is the control modules which enable students and teachers to design and construct their own control models without much difficulty.

Much work, however, is needed in the design of a programming language and teaching material.

CHAPTER 3

LEARNING THROUGH PROGRAMMING

The first section of this chapter outlines the rationale behind the methodology of learning through programming. The second and third sections review four computer languages, namely Logo, Prolog, Smalltalk and Forth. All these languages are interactive and extensible. These aspects of a computer language are particularly valuable in an educational setting (Harvey, 1984). However, these languages are fundamentally different. Logo, Prolog and Smalltalk all have some prior association with educational computing. Besides reviewing their language features, their uses in schools are also considered. Forth was originally designed as a convenient language for programming computer controlled equipment. Its suitability as a language for learning is also considered. The final section concludes that none of these languages, in their present forms, is ideal for learning control applications. It also gives some reasons why Logo is most suited for extension.

3.1 Programming aids understanding

Boden (1977) defined Artificial Intelligence as ´the use of computer programs and programming techniques to cast light on the principles of intelligence in general and human thought in particular.´ Implicit in the statement is the belief that program construction aids understanding. There are three reasons for this belief. First, in order to construct a program that performs an intelligent function it is necessary to reflect on the nature of the function to be performed. Second, a program provides a model for understanding relationships, constraints and rules of complex

systems, such as human brains. For example, in order to construct a vision, natural language, learning or expert systems program it is necessary to probe deeply into the nature of the specific domain. Third, programs are testable. They can be used to verify or to model different theories.

Papert, in his book 'Mindstorms: children, computers and powerful ideas' (1980) proposes 'to teach Artificial Intelligence to children so that they, too, can think more concretely about mental processes. While psychologists use ideas from A.I. to build formal, scientific theories about mental processes, children use the same ideas in a more informal and personal way to think about themselves.' More importantly, the thesis is that children should program computers, and that computers should not program children. This would give children 'a mastery over a piece of the most modern and powerful technology and establish an intimate contact with some of the deepest ideas from science, from mathematics, and from the art of intellectual model building.'

Programming helps a learner to focus attention and gives concrete forms to abstract ideas. With a suitable programming environment a student can engage in purposeful activities. A student constructs a program which models the solution to a problem. The program represents at least part of the student's understanding of the problem, and its correctness can be checked by running the program on a computer. Any error detected can usually be debugged and corrected.

Many students have acquired the fear of learning. They tend to associate their unsuccessful learning experiences with their

inabilities and classify themselves as good at X but not at Y. This view has the effect of stopping many students from trying to learn somthing that they think they are not good at. Programming is a potentially effective way to change one's view of success and failure. When programming a computer, one almost always gets it wrong the first time. Debugging is an important component of programming. Since making errors is the norm it is not so intimidating.

Howe (1980) draws a revealing analogy between building physical models using a kit of parts, e.g. Meccano, and building computer programs using a programming language. Just as building a physical model would help a child to understand the structure and mechanism of working machines, building computer programs would help a child to understand the mechanism of processes. A physical model that is ill-designed would collapse or the mechanical parts would cease to function. A faulty program would also cause errors when executed by a computer. The feedback information would help a student to identify faults in his own understanding and refine it. Figure 3.1 shows the relationship between learning to build physical models and learning to write programs.

Modelling	Programming
become familiar with the components of the modelling kit	become familiar with the primitives of the computer language
learn the basic operations of assembling	learn the basic operations of constructing programs
understand the structure and the mechanism of the machine being modelled	understand the problem to be solved
plan assembly sequence	plan program implementation
represent the essential structure and mechanism using the modelling kit	describe the solutions to the problem and its sub-problems in the computer language
make modification to cope with mismatch	debug and correct any errors

Figure 3.1 Relationship between building models and writing programs

A further advantage associated with learning through programming is that programming is enjoyable. Students can spend hours in front of a computer terminal without losing interest. Using the same model building analogy, Howe (1980) points out that 'a child is often more interested in grappling with the problems of assembling an object from its parts than he is in playing with the final product.'

Although there are clear advantages in learning through programming, in order that it is successfully applied there needs to be a programming language suitable for a particular application. Just as, given sufficient time and patience, one can build models of any physical structures with matchsticks, so one can write any program in machine code, but that would defeat the purpose of the

exercise.

3.2 Logo

3.2.1 The language

The computer language most associated with the programming approach is Logo (Abelson, 1980). It was first designed by Feurzeig et al (1969) for investigating the teaching of mathematics. The language is based on LISP (Winston and Horn, 1981), an Artificial Intelligence computer language. Logo is a powerful language and it is only in the past few years that it has been feasible to implement it on inexpensive microcomputers for school use. Since Logo has been available on microcomputers it has gained immense popularity. Its success owes much to the previously mentioned turtle commands extension to the original design. However, Logo is much more than turtle commands.

From the programming environment point of view, Logo has an integrated screen editor and an integrated filing system. This means that a user only needs to talk to Logo and need not learn an operating system.

Data structures and related operations

Logo has three data types: numbers, words and lists. Most implementations support both real and integer numbers and provide the full range of arithmetic and trigonometry functions. A word in Logo is essentially a string of characters. A list is an ordered set of numbers, words, and lists. The following are some examples of valid lists:

```
[THIS IS A LIST OF WORDS]
[THE EMERGENCY TELEPHONE NUMBER IS 999]
[TOM [HEIGHT [6 FEET]] [BIRTHDAY [23 JANUARY 1920]]]
```

Most programming languages, for example BASIC and Pascal, provide arrays instead of lists. There are two major differences between these two types of data structures. First, an array has a fixed size but the size of a list can shrink or grow during run time. Second, an array is uniform, i.e. each array element must be of the same type. As shown above, a list element can be any Logo data structure. A list is sufficiently general to represent and store any kind of data.

The reason that some languages provide arrays instead of lists is that, being fixed size and uniform, the location of any element in the array can be easily calculated and the content can then be accessed or updated directly. However, lists are stored in a more complicated way because the elements can be of different sizes. To access the nth element of the list, the computer has to do a search starting with the first element of the list, then find out where the second one is and so on. Since these operations are all done by the Logo system, to the programmer list manipulation is more powerful but not more difficult to use than arrays. The disadvantage is that list processing is slower.

Logo is not typed, so that a variable can be used to store any value; an integer at one point and a list or a word at another. In BASIC a variable name which ends with a dollar sign can only store a string of characters. Other variables can only store numbers. In Pascal a variable has to be declared as a specific type. Originally,

variable typing is for the convenience of the compiler to generate efficient code. There are some who would argue that variable typing is good, apart from implementation issues, because it disciplines the programmer to use variables carefully. Instead of this restriction, Logo programmers are encouraged to write small procedures which have their own local variables, which makes programs easier to write and understand. Furthermore, a programming language is much more flexible if it is not typed.

Control structures

Logo is a single-process sequential language. It allows recursive procedure calls, i.e. a procedure can call itself. Most implementations provide the control commands REPEAT and IF, with a full range of conditional tests. Since Logo is extensible and has list processing, other common control commands can be written in Logo itself. For example the WHILE command can be defined as:

```
WHILE ´CONDITION ´ACTION
  RUN :CONDITION
  IFFALSE [STOP]
  RUN :ACTION
  WHILE :CONDITION :ACTION
```

Listing 3.1 WHILE program

Then, the commands

```
MAKE ´X 10
WHILE [:X > 0] [PRINT :X MAKE ´X :X - 1]
```

would print the numbers from 10 down to 1 on the screen. However, control commands implemented in Logo are slow.

Other commands

Logo also has commands for music. The most recent extension is Sprites (Musha, 1981). They are graphical objects that can be told to carry a shape, to move at a fixed speed, and to move in a fixed direction. As an example, the commands

```
TELL SPRITE 1
CARRY       :TRUCK
SETCOLOR    :BLUE
SETHEADING 90
SETSPEED   60
```

Listing 3.2 A Sprite program

tell SPRITE 1 to carry a BLUE TRUCK and move continuously across the screen at SPEED 60. The speed of a sprite is an arbitrary unit. A student can define his own shapes for the sprites and many sprites can be made to move across the screen simultaneously. The sprites provide a simple way for students to write animation programs.

A simplified view of Logo and the curricula may be presented in diagram form as in figure 3.2.

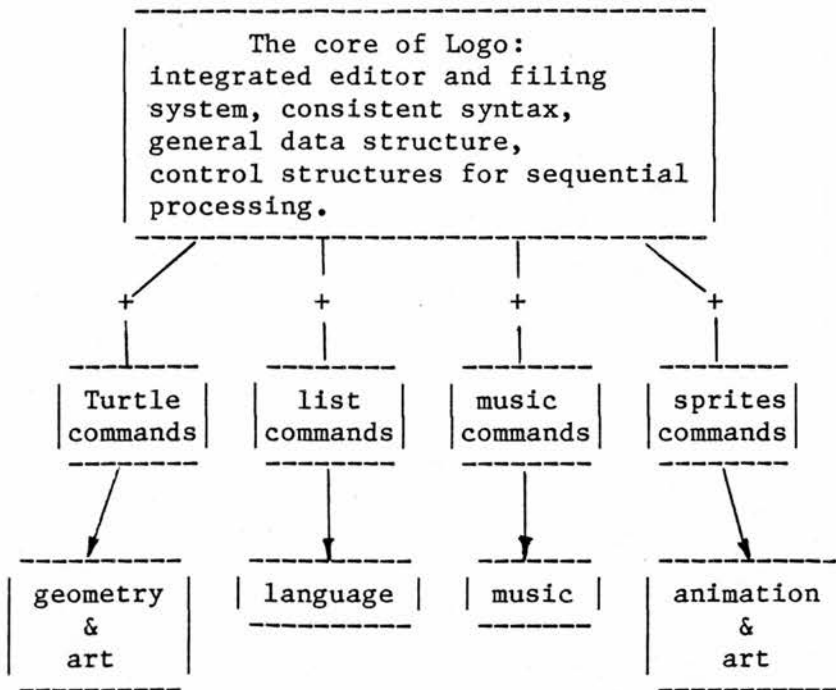


Figure 3.2 Logo and the curricula

3.2.2 Logo programming

Logo programming is usually seen as a cyclic process: planning, coding, debugging, planning, coding, debugging and so on, until the execution of the program meets the intention. In an interactive system such as Logo, and given the different abilities and experiences of novice programmers, the programming activities do not necessarily fall neatly into this pattern. Solomon (1982) described three styles exhibited by students programming in Logo: the planner, the macro-explorer and the micro-explorer. A planner would carry out a careful analysis of the problem and then build procedures from a structured plan. Not necessarily with the aid of a plan, the macro-explorer would build procedures and examine their effects. Through exploring, he would make new concepts his own and extend his control

over the programming environment. A micro-explorer is the most timid programmer. He would only use a small set of primitive commands that are familiar to him. Papert et al (1979) described how one student only used the basic Turtle commands in her work. For inputs to the commands, she used mostly 30, 60 and 90. To make the turtle move formard 120, she would use the commands FORWARD 90 followed by FORWARD 30.

Some students, but not all, fit clearly into one of the categories above. Following Noss (1983), it is appropriate to interpret different styles as different modes. Micro-exploring can be seen as 'making sense of a new idea', such as learning the syntax of a command and appreciating the function of the command. Macro-exploring is extending a new idea, by incorporating it in a new procedure or by experimenting with the idea. Exploring helps to link new ideas with existing knowledge. Planning is problem solving in a goal-directed way. When programming, a novice programmer may (and some often do) switch between these different modes. Interaction between the different modes is shown in figure 3.3 (adapted from Noss, 1983). This model of programming is useful to a teacher for deciding when and how to help a student. For example, when a student is making sense of a command he should be given an explanation and time to try it out until he feels confident with it. When he is exploring an idea he may need suggestions on what may be tried or where to focus his attention. When the student is planning a solution, a teacher may help by discussing with him the different ways that a problem may be solved.

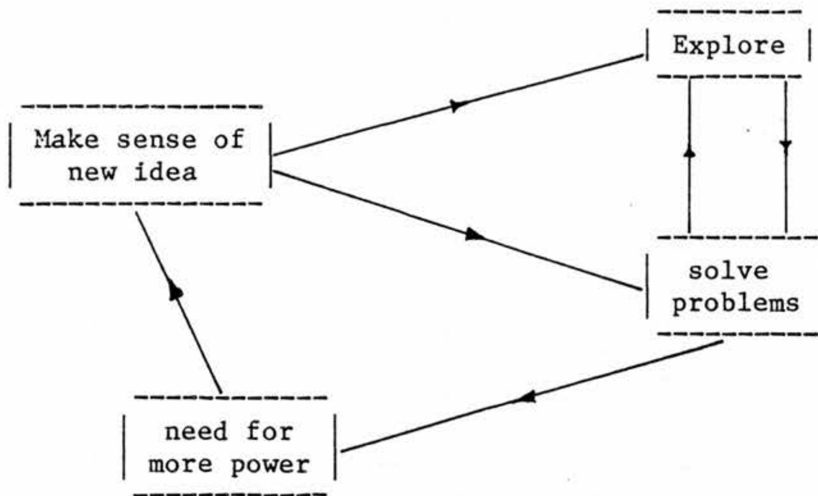


Figure 3.3 Interactions between programming modes

Although students can do interesting things with Logo from the very first time they use it, there are features in Logo that require some time to get used to. The use of single quote "´", and colon ":" to distinguish name from value does cause confusion among beginners. Many students have difficulties in understanding the elegant and powerful idea of recursion, i.e. a procedure calling itself. Kurland and Pea (1983) found that some students misunderstood recursion as looping - a jump back to the beginning of the procedure. Though this model is faulty, it adquately explains tail recursion - a procedure calls itself in the last command of the procedure. The problem is understanding non-tail recursive programs.

3.2.3 Evaluation studies

Logo has been suggested and used for teaching many school subjects. For example: physics (diSessa, 1980), biology (Abelson and Goldberg, 1977), English language (Sharples, 1980; Rowe, 1976) and music (Bamberger, 1972; Bamberger, 1979). However, most formal

evaluation studies have been done in the context of teaching mathematics to school children. The result is favourable.

Using the pre-post test method, Milner (1973), Howe et al (1980) Hartley (1980) and Howe et al (1982) have found that the scores of the experimental groups improved relative to the performance of the control groups. Hartley (1980) and Howe et al (1980) also reported that the students who learned mathematics through programming had increased their ability and willingness to discuss mathematics.

du Boulay (1978), working with trainee primary school maths teachers who found maths difficult, provides evidence that some basic maths concepts can be learned effectively through programming. On the other hand, he also made clear that programming does not fit in well with all maths topics. An example given by du Boulay is representing fractions as pie charts. His students concentrated in constructing the programs to do the drawing and did not gain understanding of fractions. This demonstrates that programming tasks have to be relevant; helping the students to focus on the concepts being learned and demanding little programming skills.

Because there is a clear relationship between programming and problem solving, some advocate that learning to program a computer can enhance a student's intellectual functioning. The idea is clearly expounded in Papert's writing (1980). The following extracts help to examplify the point:

> 'Indeed, the role I give to the computer is that of a carrier of cultural "germs" or "seeds" whose intellectual products will not need technological support once they take root in an actively growing mind.' p.9

> '.... children who had learned to program computers could use

very concrete models to think about thinking and to learn about learning and in doing so, enhance their powers as psychologists and as epistemologists.' p.23

'.... through these experiences these children would be serving their apprenticeships as epistemologists, that is to say learning to think articulately about thinking.' p.27

Several studies have been done to examine the claim that programming improves a student's problem solving skills. The first one was by Statz (1973). She taught programming to 16 students of the age 9-11 over a one year period. She hypothesized that this group would do better on a set of four problem solving tasks than a control group who learned no programming. The outcome was that the experimental group did significantly better in only two of the tests. However, there are criticisms of the way she marked the tests (Weyer and Cannara, 1975) and there are doubts about the validity of the tests as they do not measure the kind of problem solving skills that are likely to be learned through programming.

Papert et al (1979) also taught Logo programming to sixteen sixth graders (age 11-12) over a period of six weeks. These students had between 20 and 40 hours of hands-on experience with the computers. The analysis of the students' work shows that the students were engaged in extensive problem solving activities and that they had different programming and problem solving behaviours. However, it provides no evidence that the problem solving skills gained in programming are transfered to other non-programming tasks.

More recently, Pea & Kurland (1983) taught programming to two classes of 25 students (age 8-9, 11-12) for one year. Since the development of planning abilities is one major predicted benefit of learning to program, they developed a transfer task for assessing

children's planning (Pea and Hawkins, 1983). The task was given twice, early and late in the school year, to eight children in each of the two programming classes, and to a control group of the same number of same-age students from the same school. The experimental groups did not display greater planning skill than the control groups.

Undoubtedly some would argue that the cognitive benefits of programming would either be manifested only if a student is subject to a much longer exposure to programming, or revealed only in later years.

Based on two general findings in cognitive science and his own observations, Pea (1983) gives three reasons why the claim made for the cognitive benefits of learning to program is doubtful. First, the 'transfer of problem-solving strategies between dissimilar problems or problems of different content, is notoriously difficult to achieve even for adults'. Second, even computer science students, who have had several thousand hours of programming experience, have great conceptual difficulties in understanding how simple programs work. The programming experience and skills they had acquired do not help them to solve problems related to programming. Third, the context-free problem solving skills such as planning and debugging are not necessarily developed through programming. Pea observed that his students do very little pre-planning in their programming activities. Furthermore, if the outcome of a program is not satisfactory his students would change the goal and restart, rather than revise the program.

The evidence accumulated so far, though scanty, shows that programming can be used to enrich classroom activities and help students to focus their attention and gain insight into concepts that are directly relevant. It is doubtful that general problem solving skills can be automatically acquired through programming.

3.2.4 Teaching methods

There are two contrasting views on how Logo should be introduced into the school classroom. One view is that the programming activities should fit into the existing classroom practice. The teaching material should be structured. The student is asked to write programs that model a particular process or concept with the aim of promoting deeper understanding. The other view favours open-ended investigation. The student is encouraged to create his own projects and to explore his own ideas. By providing a suitable programming environment, he may learn through self-discovery rather than organised instruction. The latter view is supported particularly by those who advocate that programming helps to develop general problem solving skills.

The previously mentioned studies by Howe et al (1980; 1982) and Hartley (1980) were carried out in the structured fashion, particularly the studies by Howe and his colleagues. Though the result is favourable, the structured approach has been bluntly criticised as antithetical to the Logo philosophy (Kelman, 1983).

The studies by Papert et al (1979) and Pea et al (1983) were carried out in the open-ended fashion. Papert made no comments on the effectiveness of the teaching method. Pea concluded that 'we have

deep doubts, based on a series of empirical studies over an 18-month period, that the Logo ideal is attainable with its discovery-learning pedagogy.′ In Statz′s study (1973), three teaching strategies were used. They varied in the degree in which projects were defined by the students and the point at which Logo concepts were introduced to the students. She found that a certain degree of structure in the teaching of Logo is required.

Learning through discovery is idealised rather than practical. It assumes that a student

(1) can develop his own projects; this requires initiative and creativity.

(2) can appreciate the complexity of the project he is undertaking; this requires some background knowledge and understanding.

(3) is sufficiently motivated to pursue the project for a long period of time in order to benefit from it.

It also assumes that the programming language is sufficiently general to support the kind of projects that a student is interested in.

A teaching-learning situation should take into account the nature of the learning task; the advantages and limitations of the programming environment; and certain characteristics of learners, such as their prior knowledge, level of ability and motivation. As observed by Pea, most students will not rethink their ideas, even when they see that their programs are wrong. Structure is necessary. It provides a framework and a collection of ideas that a teacher can follow or adapt. Teaching material should be categorized into different levels of difficulty. Worksheets can be used to help a

student to learn the basic ideas quickly and provide suggestions of what might be tried next. The important thing is that a teacher should have an understanding of the likely mistakes that the students would make, why they make them and should provide help when appropriate. The model of programming described in section 3.2.2 is a helpful guide line.

3.3 Other languages

This section reviews three other computer languages, namely Prolog, Smalltalk and Forth. The former two languages have some association with educational computing and Forth was designed as a language for programming computer controlled equipment. Their suitability as languages for learning are considered.

3.3.1 Prolog

The name 'Prolog' stands for PROgramming in LOGic. It is a computer language based on Horn Clause predicate logic (Kowalski, 1974). Prolog has been chosen by the Japanese as the core programming language for the next generation of computers. In Europe, Prolog is widely used in Artificial Intelligence research for building expert systems and natural language front ends. Kowalski (1984) and Ennals (1984) advocate that logic is a good programming language for teaching children. Recently, the Irish Department of Education has decided to make Prolog available in all its secondary schools. A number of other British education authorities are also showing interest.

3.3.1.1 Distinctive feature of Prolog

The distinctive feature of Prolog is that it allows programs to be written in declarative form rather than in algorithmic form, i.e. telling the computer what to do rather than how to do it. A program in Prolog consists of statements of facts and rules. Typically a statement is of the form:

```
Conclusion.

or

Conclusion
        if Goal1, Goal2, ..., GoalN.
```

which can be read as: if Goal1, Goal2, ... and GoalN are true, then the Conclusion is true.

Listing 3.3 is a simple Prolog program written in micro-Prolog (McCabe, 1980-81), the most widely used implementation in schools today.

```
(father-of Jack George)
(father-of Tom Bill)
(father-of Bob Jack)
(father-of Bernard Tom)
((grandfather-of X Z) (father-of X Y) (father-of Y Z))
```

Listing 3.3 Family relation program

In listing 3.3, the first four statements are facts stating that Jack is the father of George, Tom is the father of Bill, Bob is the father of Jack and Bernard is the father of Tom; the last statement is a rule describing the grandfather relationship: X is the grandfather of Z if X is the father of Y and Y is the father of Z. Note that the program is incorrect in some sense because X can be the grandfather of Z if X is the father of Y and Y is the mother of Z.

If a Prolog system has these statements stored in it, one can ask it questions like 'Who is the father of Jack?' and 'Who is the grandson of Bernard?' These questions are represented in Prolog as

```
(father-of X Jack)
(grandfather-of Bernard Z)
```

respectively. Prolog would reply with the answers X is Bob and Z is Bill.

Another example that shows the declarative power of Prolog is the program for appending two lists:

```
(append () X X)
((append (X | Y1) Y2 (X | Y3)) (append Y1 Y2 Y3))
```

Listing 3.4 Append program

The program appends the lists in the first two arguments and returns the result in the third argument. The first clause states that if the first list is empty then any list appended to it is the same list. The second clause states that

(1) the first element of the first list X is always the first element of the final list.

(2) the tail of the final list, Y3, is the second list, Y2, appended to the tail of the first list, Y1.

The program is a description of the relationship between the lists to be appended and the appended list, rather than a description of how two lists are appended.

3.3.1.2 Prolog in schools

In schools, Prolog is used, typically, as a data base query language. The content of the data base can be related to any subject; for example: the history of a village and the geographical information avout a country. The students learn by finding out and providing facts and rules to the system. Alternatively the information could be supplied by the teachers and the students then conduct investigations by querying the system. Some teaching material for using micro-PROLOG has been published (Ennals, 1982). Efforts have been focused on evaluating and improving the user friendliness of the interactive facilities in micro-PROLOG (Sergot, 1984; Weir, 1982). The front end (called SIMPLE) to micro-PROLOG allows students to input data base statements or queries in a form closer to natural language. For example the rule that describes the grandfather relationship and the query 'Who is the grandfather of George?' can be represented as

```
X grandfather-of Z
        if X father-of Y & Y father-of Z

which(x : x grandfather-of George)
```

respectively.

So far, there is no evaluation of the educational benefits of programming in logic.

3.3.1.3 Programming in Prolog

Ideally logic programs should be read declaratively and understood without recourse to the behaviour they invoke inside a machine. However, Prolog programs typically cannot. It is because

Prolog uses a very simple depth-first proof procedure. It means that

(1) the ordering of the goals within a statement is important;

(2) the ordering of the statements for a predicate is important;

(3) sometimes extra-logical primitives have to be used.

As an illustration, consider the program SUM_TO (listing 3.5) written in SIMPLE syntax.

```
SUM_TO(1 1)
SUM_TO(x y) if
        x1 = (x - 1) &
        SUM_TO(x1 y1) &
        y = (x + y1)
```

Listing 3.5 SUM_TO program

The program expects a goal of the form SUM_TO(N X), where N is assumed to be a positive integer and instantiates X to the sum of the numbers from 1 to N. Given the goal SUM_TO(3 X) X would be instantiated to 6 since 6 = 1 + 2 + 3. However, by swapping the order of the two statements, i.e.

```
SUM_TO(x y) if
        x1 = (x - 1) &
        SUM_TO(x1 y1) &
        y = (x + y1)
SUM_TO(1 1)
```

which does not change the logic of the program, the program would loop infinitely.

Although the SUM_TO program is logically correct and seems to work it would easily cause an infinite loop when used as a part of a larger program. Consider the program SMALL_SUM (listing 3.6), written to find out whether the result of SUM_TO is less than 12:

```
SMALL_SUM(X) if
        SUM_TO(X Y) &
        Y LESS 12
```

Listing 3.6 SMALL_SUM program

The logic of the program is correct, but if the result Y is greater than or equal to 12 Prolog would loop infinitely. For example, the goal SMALL_SUM(4) would succeed but the goal SMALL_SUM(5) would cause an infinite loop. To correct the error an extra-logical primitve called ´cut´, written as ´\´, has to be used in the definition of SUM_TO to control the backtracking mechanism in Prolog. The correct Prolog defintion of SUM_TO should be

```
SUM_TO(1 1) if \
SUM_TO(x y) if
        x1 = (x - 1) &
        SUM_TO(x1 y1) &
        y = (x + y1)
```

Listing 3.7 Modified SUM_TO program

It is very difficult for a novice to form a model of how a Prolog system works. What stories to tell a Prolog student is a major research topic (Bundy, 1983). Some advances has been made in the development of debugging aids (Byrd, 1980). There is currently a lot of research work going on to improve the control aspects of Prolog (for example see Clark and McCabe, 1982; Clark and Gregory, 1983; Naish, 1982; Naish, 1983 and Shapiro, 1983)

Despite the apparent simplicity of using Prolog in a secondary school classroom the extent to which Prolog can be used beyond data base and query manipulation remains to be seen.

3.3.2 Smalltalk

Smalltalk was developed by the Learning Research Group at the Xerox Palo Alto Research Center in California. The conception of the language was strongly influenced by

(1) the Logo philiosophy that children can program and will benefit from it;

(2) the vision that in the near future young children will have access to very powerful computing facilities. It was the group's aim to develop a computing environment for the future generation of personal computers for children (Kay, 1977).

The language is designed around a single concept - that similar objects can be grouped into more general classes. Every entity in the system is considered as an object and each object is an instance of a class. These objects can receive messages, which tell them to do something, remember something, recall something, or send some messages to other objects. Sprites in Logo may be considered as a class of system objects that can respond to the messages CARRY, SETCOLOR, SETHEADING, and SETSPEED. In Smalltalk users can create their own classes, objects and messages.

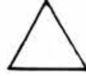
To illustrate the class concept in Smalltalk, the following example (listing 3.8) defines a class TRIANGLE and shows how different triangular objects of that class can then be created and manipulated. The program is not written in strict Smalltalk syntax.

```
CLASS NAME;

        @define a new class TRIANGLE
        TRIANGLE.


CLASS MESSAGE;

        @NEW creates a new instance of class TRIANGLE.
        @The new instance is given a default 'shape' and
        @location.
        NEW: LOCATION <- CENTER, ANGLE <- 120, LENGTH <- 100.


INSTANCE MESSAGES;

        @SHAPE tells an instance to draw its shape on the
        @screen
        SHAPE: GOTO LOCATION, PENDOWN,
               1 TO 3 DO (FORWARD LENGTH, TURNRIGHT ANGLE).

        @SHOW tells an instance to draw its shape on the
        @screen in black
        SHOW: PAINT BLACK SHAPE.

        @ERASE tells an instance to draw its shape on the
        @screen in BACKGROUND colour
        ERASE: PAINT BACKGROUND SHAPE.

        @GROW tells an instance to change its size.
        @This is done by changing it side length by the amount
        @specified by the user.
        GROW []: ERASE, LENGTH <- LENGTH + [], SHOW.

        @TURN tells and instance to rotate.
        TURN []: ERASE, TURNRIGHT [], SHOW.
```

Listing 3.8 TRIANGLE program

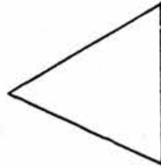
The following example (listing 3.9) shows how different triangular objects can be created and manipulated:-

```
@create a new instance of class TRIANGLE, named JACK
TRIANGLE NEW NAMED 'JACK'!
```

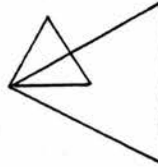


```
@send a message to JACK, tell it to turn 30 degree right
JACK TURN 30!
```

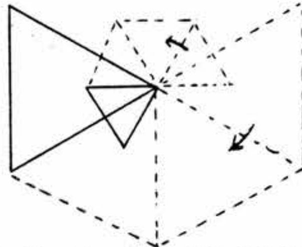
```
@send a message to JACK, tell it to increase the length of its
@side by 120 units
JACK GROW 120!
```



```
@create a new instance of class TRIANGLE, named BILL
TRIANGLE NEW NAMED 'BILL'!
```



```
@repeat the following 3 times:
@send a message to JACK, telling it to turn right 60 degrees,
@then send a message to BILL, telling it to turn left 60 degrees
1 TO 3 DO (JACK TURN 60. BILL TURN -60)!
```



Listing 3.9 Manipulating TRIANGLEs

The advantages of the class concept are:

(1) An object is a computational entity that combines both the data and the operations that are allowed to be performed on the data, therefore objects subsume procedures, functions, and all kinds of data structures.

(2) It is conceptually clean, having a natural and clear meaning.

(3) It is not possible to change any data that is local to an object without sending that object a message requiring such an operation, thus providing integrity.

(4) It is suitable for applications that involve modelling and manipulation of abstract or physical objects.

The class concept described so far is similar to abstract data type, as implemented in other programming languages such as Concurrent-Pascal (Brinch Hansen, 1975). Each object belongs to only one class. There is no intersection between classes. Pictorially it can be represented as figure 3.4. In the figure, a rectangular shape represents a class and an asterisk represent an instance of a class.

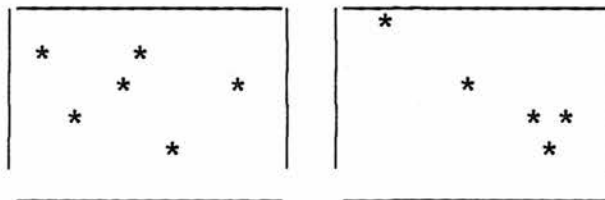


Figure 3.4 Mutually exclusive classes

Smalltalk also provides a subclass mechanism. Figure 3.5 illustrates it. A subclass specifies that its instances are the same as those of another class, called its superclass, except for the

differences that are explicitly stated. Each subclass has one superclass and many subclasses may share the same superclass. A subclass is in all respects a class and can therefore have subclasses itself. This subclass mechanism is useful for describing different classes of objects that have essentially the same features but differ in some details.

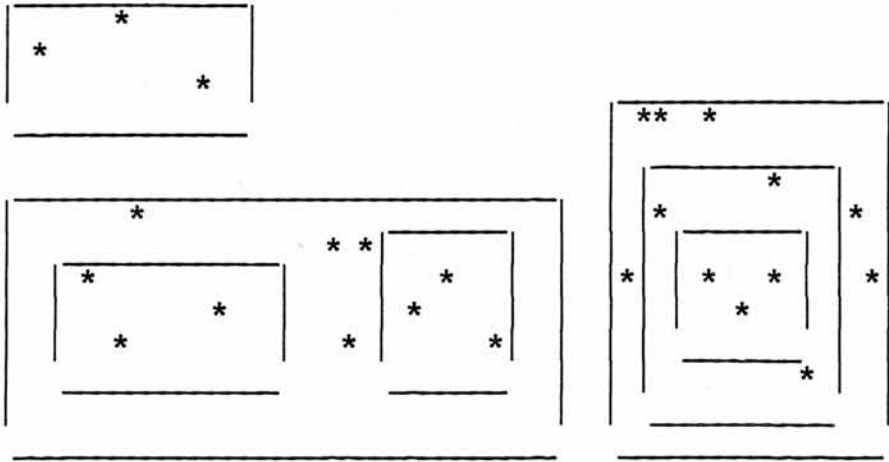


Figure 3.5 Hierarchical classes

Smalltalk has been used in an experimental study with a group of specially gifted children (Goldberg, 1977). The subject material taught included computer simulation methods, graphic techniques, geometry and animation. The course work was designed to take full advantage of the graphics capability of Smalltalk. Careful consideration was given to how Smalltalk should be introduced to pupils for modelling purposes. It is difficult for the uninitiated to decide what objects are required and what messages the objects are to receive. The teaching strategy developed was to provide interesting pre-programmed classes so that the students could use,

modify and combine them. Because the students were talented and had previous programming experience they were able to produce many interesting pictures and simulations.

Unfortunately, the aim of the development of Smalltalk has drifted. It is now more concerned with software development for professionals (Goldberg and Ross, 1981). It is interesting to note that the name of the research group has also changed from Learning Research Group to Software Concepts Group. The major contributions of Smalltalk are the concepts of windows and menus. They emphasise multiple screen displays and the use of a pointing device for selecting commands, rather than using the keyboard. Perhaps Smalltalk now plays a new role in education; it is a language for implementing educational software. It was used in the development of Thinglab (Borning, 1979) and TRIP (Gould and Finzer, 1981).

3.3.3 Forth

Forth (De Grandis-Harrison, 1983) was invented around 1969 by Charles Moore. It was originally created as a convenient means of controlling equipment by computer. It has the advantages that it executs quickly, requires little memory for program storage, and encourages structured programming by breaking down a program into small sub-procedures.

The distinctive features of Forth are that it uses post-fix notation and that the system stack can be manipulated directly by a programmer.

Post-fix notation

A computer language that uses post-fix notation requires the operands to be specified before their operators. For example the arithmetic expression

2 + 3 - 4

would be written in post-fix form as

2 3 + 4 -

A more complicated expression

(2 + 3) * 5

could be written either as

2 3 + 5 *

or as

5 2 3 + *

If Forth had turtle graphics the command FORWARD 100 would be written as: 100 FORWARD.

Stack manipulation

Stack is a last-in first-out (LIFO) data structure where the value most recently placed on the stack is most accessible. It is similar to the pop-up pile of plates one might see in restaurants. If a plate is placed on the top of the pile it moves down until the new plate is at the counter level. If a plate is removed the pile rises so that the plate which was underneath becomes the new top of

the pile.

Most high-level languages use one or more stacks for their internal operations, for example for storing intermediate results of arithmetic calculations. However, a Forth programmer manipulates the system stack directly. As an illustration, consider the evaluation of the expression

```
2 3 + 4 -
```

It first tells the system to put the value 2 on the stack and then the value 3. At this point the stack looks like

```
3       <- top of stack
2
```

The system is then told to add the top two values of the stack, which is 3 and 2. The system removes the values and then put the sum on the top of stack. At this point the stack looks like

```
5       <- top of stack
```

The system is then told to put the value 4 on the top of stack, thus the stack looks like

```
4       <- top of stack
5
```

The minus sign tells the system to subtract the value on the top of stack from the one that is underneath it, so at the end of evaluating the expression the result 1 is found on top of the stack. Forth provides many commands for stack manipulation. For example:

```
DROP    remove top of stack item
DUP     duplicate the top of stack item
```

SWAP   exchange the top two items.

As an example, consider writing a COUNT program in FORTH. The definition

```
COUNT
DO I . LOOP
```

Listing 3.10 COUNT program

is a first approximation. The command ´I´ places the value of the current loop index on the top of stack. ´.´ means print and remove the value on the top of stack. The command

```
6 0 COUNT
```

prints the sequence 0 1 2 3 4 5 on the screen. Notice in this definition the upper limit of COUNT has to be given before the lower limit. The sequence printed is only up to upperlimit - 1. To overcome them COUNT may be modified to

```
COUNT
1 + SWAP DO I . LOOP
```

Listing 3.11 Modified COUNT program

The command

```
0 6 COUNT
```

now prints the sequence 0 1 2 3 4 5 6. Writing programs in FORTH requires substantial understanding of stack manipulation. The final programs are far from comprehensible.

3.4 Learning control applications through programming

As indicated in section 3.2.3, the prevailing evidence suuggests that programming is a good way for a student to gain insight into specific aspects of a subject being studied, especially those aspects concerned with processes. There are good reasons to believe that the approach is also particularly suited for control applications:

(1) Working with models of control systems is compelling.

(2) It requires no prior knowledge of electronics. Programming is directed toward producing descriptions of control processes; a student might come to understand the general principles underlying the control system.

(3) A moving device provides visual feedback of the control process in action; the student programmer is more able to realise the discrepancy between the effect and his intention.

As explained in Chapter 1, a course in control applications should cover a wide range of control devices, so that they represent a range of different applications and the programming tasks make use of different control concepts. With these requirements it is appropriate to apply the structured teaching method. The essential requirement is a computer language containing commands that enable the control processes to be conveniently described in program form.

One basic requirement of a control language is that it should allow easy I/O handling, since controlling a device or process requires that data be received from, and sent to, external components. In control applications, it is common to talk about objects (the device being controlled and its components) - what they are for and what they can do. When writing a control program it is

convenient to model what the program does in terms of the components of the device being controlled. For example, if a motor drives a cart, a high level description of how the cart works could be 'the cart moves forward when the motor turns clockwise; the cart moves backward when the motor turns anticlockwise; the cart stops when the motor stops.' It is appropriate to use the metaphor of sending commands to objects telling them to do something. Therefore, a programming language for control applications should also provide

(1) commands for manipulating objects that can be likened to real world things, especially components common in control devices;

(2) a convenient notation for addressing individual components. This is to make sure that a command is sent to the right component, because there may be a number of components of the same class (type) used on a control device.

This view is in accord with the growing trend in the development of computer languages for real time applications - incorporation of abstract data types (for examples Concurrent Pascal (Brinch Hensen, 1975), Modula (Wirth, 1977) and Ada (Goos and Hartmanis, 1983)). As described in section 3.3.2, an abstract data type is similar to the class concept found in Smalltalk. The major difference is that abstract data types do not provide the subclass mechanism.

Another important feature of a control language is that it should have powerful control structures. The term control structure refer to 'both implicit global interpretation rules for programming languages and explicit control operations' (Fisher 1972). Sequencing, repetition, conditional statements and hierarchical procedure calls are sufficient for most non-control applications.

However, it is essential for a control language to have a multi-programming capability. For example, to make two stepping motors turn 30 steps simultaneously, it would be convenient if the language allowed the programmer to write a program similar to

```
MOTOR 1 TURN 30 STEPS in parallel with MOTOR 2 TURN 30 STEPS
```

However, if the language allows only one process, a sequential algorithm that tells the computer to switch between the two motors:

```
REPEAT 30 TIMES
MOTOR 1 TURN 1 STEP
MOTOR 2 TURN 1 STEP
```

has to be used. This solution obscures the logic of the program. Furthermore, a control program usually describes a process that continuously checks for occurrences of a number of signals or events and then responds to them as they arrive. Sometimes, several components or events might require attention simultaneously. To write a control algorithm using a single-process sequential language is extremely difficult. Consider translating the algorithm

```
WHENEVER (signal 1) DO (action 1; action 2; action 3)
        in parallel with
WHENEVER (signal 2) DO (action 4; action 5; action 6)
```

into a single process sequential program. It is therefore essential that a control language provides control structures that are convenient for describing event handling, such as WHENEVER, and for multi-programming.

Finally, a control language should be speed efficient. This is to ensure reliability and acceptable performance. If a control program is written correctly but its response time is slow, it may

miss some signals or events and fail to take the specified actions.

The characteristic which makes writing a program for control applications different, and difficult in comparison with writing programs for other applications, is the control flow of the program. For most applications, the run time behaviour of a program, even of one using recursion, can be understood using a sequential model. Commands are obeyed one at a time in a linear and deterministic way. Furthermore, the programmer does not have to worry about the time it takes for a computer to obey a command. From the correctness point of view, the length of time that a computer takes to draw a line or to find the nth element of a list is irrelevant. However, in control applications the control flow model is more complicated and timing is essential. Consider the following control program:

```
REPEAT 1000 (
        IF (signal 1) DO (procedure 1)
        IF (signal 2) DO (procedure 2)
)
```

The run time behaviour of the program depends on when signals 1 and 2 are on and how long they stay on. It also depends on how long it takes to execute procedures 1 and 2. Suppose, for example, signal 1 was on and the computer was executing procedure 1. In the meantime, signal 2 was on for a short while but the computer was not checking for it because it was busy executing procedure 1. When the computer checked for signal 2, it was off again. So, the control flow of a control program can be said to be non-deterministic. The programmer has to reason about timing and whether different events are likely to happen at the same time. He needs to develop a model of parallelism.

To recap, the aim of teaching control applications is to help a student to understand how a control system works. In A.I. terms, this type of knowledge is called ´procedural´, as oppose to ´factual´. It is best captured by procedures that describe algorithms explicitly. Therefore the algorithmic programming languages are more appropriate than the declarative ones. This means that Prolog as yet is not entirely suitable. However, there is also no algorithmic programming language that meets the above stated requirements . Instead of a new language designed from scratch or a sophisticated language like Smalltalk, Logo is a good language which can be suitably extended. A further advantage of using Logo as the base language is that it is already being used in schools and is being accepted by teachers and students.

CHAPTER 4

CONCURRENT-LOGO

Concurrent-Logo (Chung, 1984) is an extension of the programming language Logo. The aim of the extension is to provide a suitable programming language for secondary school students to learn control applications. The extended facilities include commands for detecting signals, commands for actuating switches and stepping motors, also multi-programming and guard facilities (see section 4.6).

This chapter explains the overall design philosophy of the extension and describes the additional facilities with examples of their use. The last section also describes two other implementations of Logo, namely Control-Logo and Nimbus Logo, both of which are related to Concurrent-Logo.

4.1 Design

There were two goals for the design of Concurrent-Logo:

(1) to provide a suitable notional machine for students to think about control applications and to talk about them. This meant designing suitable programming language primitives and developing a metaphor that would help students to solve problems.

(2) to maintain the virtues of Logo, such as being interactive and extensible. This meant making sure that an interpreter could be written for the extended language.

4.1.1 Object

The design of Concurrent-Logo is based on an 'object' metaphor. Concurrent-Logo provides three classes of system objects: SWITCH, RECEIVER and STEPPING MOTOR. Each class of object has a set of commands associated with it (described in section 4.2). To keep Concurrent-Logo extensible, facilities are provided for users to define new classes of object.

An infix notation is chosen for separating the name of an object from the command that is for the object. For example, the commands

```
MOTOR 1 ! TURNC 30
MOTOR 2 ! TURNA 30
```

mean: tell MOTOR 1 to TURN Clockwise 30 steps and tell MOTOR 2 to TURN Anticlockwise 30 steps respectively. The exclamation mark is the syntax marker for separating object names from commands. It is analogous to telling a person to do something or asking a person for something:

John ! Open the door,

Bill ! Help,

Steve! What is the time?

The object metaphor also applies to commands which are normal Logo primitives or procedures. It treats the Logo system as the default receiver of a command. Therefore the object metaphor for an ordinary command is 'tell the computer to do something'. For example, the command

```
PRINT ADD 2 3
```

means tell the computer to print the result of adding 2 and 3.

In Sprite-Logo there are many graphical objects of class sprites and they all respond to the same set of commands. The TELL command is used to inform the system of the current receiver(s) of sprite commands. Once a TELL command is executed, all subsequent sprite commands are send to the specified sprite(s) until another TELL is executed. TELL is advantageous if

(1) the same command is to be sent to many objects. For example the commands

```
TELL [OBJECT_1 OBJECT_2 OBJECT_3]
DO_SOMETHING
```

would send the command DO_SOMETHING to OBJECT_1, OBJECT_2 and OBJECT_3.

(2) many commands are to be sent to the same object, before switching to another one. For example

```
TELL OBJECT_1
DO_1
DO_2
DO_3
TELL OBJECT_2
DO_4
DO_5
DO_6
```

However, in control applications the above cases seldom arise. If TELL is used in Concurrent-Logo, the commands for making MOTOR 1 turn clockwise 30 steps and MOTOR 2 turn anticlockwise 30 steps would be

```
TELL 'MOTOR 1 TURNC 30
TELL 'MOTOR 2 TURNA 30
```

The exclamation mark notation has two advantages over the TELL

command. First, it requires less typing. Second, more importantly, it is meaningful when used to ask an object to return some information. It is ambiguous to ask, 'Tell Steve what is the time? ' However, we do naturally ask, 'Steve ! What is the time?'

4.1.2 Multi-programming

As explained in the previous chapter, multi-programming facilities are an important part of a programming language for control applications. Concurrent-Logo allows commands to be executed in parallel. For example the command

```
PROCEDURE_1 // PROCEDURE_2
```

would start PROCEDURE_1 and PROCEDURE_2 running at the same time. The parallel bar '//' is used to separate commands that are to be executed in parallel. Another syntax marker, semicolon ';', is also introduced for separating commands that are to be executed in sequence. For example the command

```
PROCEDURE_1 ; PROCEDURE_2
```

means execute PROCEDURE_1 then PROCEDURE_2. This is for consistency: commands are separated by markers.

4.2. I/O handling

There are system-defined objects for input and output. For output these are: SWITCH, MOTOR; for input: RECEIVER.

4.2.1 Output

SWITCH

The prototype implementation can manipulate up to 8 switches. In the prototype they are identified by subscripts: SWITCH 1, ..., SWITCH 8. A SWITCH can be told to

(1) turn itself ON
(2) turn itself OFF
(3) return its STATE.

The commands are

```
SWITCH N ! ON
SWITCH N ! OFF
SWITCH N ! STATE
```

where N is the number of the SWITCH.

A SWITCH can also respond to a more sophisticated command of the form: ONUNTIL condition. This command tells a SWITCH to switch itself on, and to switch itself off automatically when the specified condition becomes true. For example, if a heater is connected to SWITCH 1, the command

```
SWITCH 1 ! ONUNTIL GRE? TEMP 25
```

would switch the heater on until the room temperature is greater than twenty-five degrees Celsius. Note, TEMP is a user-defined procedure that returns an integer value.

The SWITCH commands are suitable for components which are usually turned on and off using hardware switches, e.g. lights and bells.

MOTOR

Programming stepping motors at a low level is a difficult task. It requires some understanding about stepping motors, the output port of the computer and the sequence of stepping patterns that drive the motors. In Concurrent-Logo, stepping motors are recognised objects. They are identified as MOTOR 1, ..., MOTOR 6. A MOTOR can be told to turn clockwise or anticlockwise. The forms of the commands are

```
MOTOR N ! TURNC M
MOTOR N ! TURNA M
```

where N is the number of the motor and M is the number of steps.

Associated with each MOTOR is a variable named COUNT. When the system is initialised each COUNT variable is set to 0. Whenever a MOTOR turns clockwise the value of its COUNT variable is automatically incremented by the number of steps turned; whenever a MOTOR turns anticlockwise the value of its COUNT variable is automatically decremented by the number of steps turned. The command

```
MOTOR N ! COUNT
```

will return MOTOR N's COUNT value. The value indicates the number of steps that the motor has turned relative to its starting position.

4.2.2 Input

RECEIVER

Receivers are system objects for detecting signals from switches which a device sends to the computer. The receivers are identified as RECEIVER 1, ..., RECEIVER 8. If the command STATE is sent to a

RECEIVER, it returns the value of its current state: ON or OFF.

A more sophisticated use of a RECEIVER is to ask it to keep count of the number of times that the input signal has changed state. The commands

```
RECEIVER N ! KEEPCOUNT
RECEIVER N ! COUNT
RECEIVER N ! CLEARCOUNT
```

will tell RECEIVER N to keep count, return the value of count and clear the count value respectively.

The RECEIVER commands are suitable for detecting signals from two-state switches.

4.3. Control structures

Two control commands are introduced in Concurrent-Logo. They are called FOREVER and WHENEVER.

FOREVER command

The FOREVER command is very simple. It is a loop without a stopping condition. The command

```
FOREVER (PRINT [THIS IS FUN])
```

prints the sentence 'THIS IS FUN' on the screen until the ESC (escape) key is pressed to interrupt it.

The command is ideal for control programs that usually have no specific stopping condition. For example, a program for controlling a lift would just detect requests and move the lift accordingly. The program would continue until the user interrupted it.

WHENEVER command

The WHENEVER command has the syntax:

```
WHENEVER <condition> ( <action to be taken> )
```

It means: take the specified action whenever the condition is true. Semantically it is equivalent to

```
FOREVER ( IF <condition> ( <action to be taken> ) )
```

For example, if a heater is connected to SWITCH 1, the commands

```
SWITCH 1! OFF;
WHENEVER LESS? TEMP 20 (SWITCH 1 ! ONUNTIL GRE? TEMP 25)
```

will first switch the heater off, then whenever the temperature drops below 20 degrees the heater will be switched on and when the temperature rises above 25 degrees the heater will be switched off automatically.

4.4 User defined objects

A user defined object is similar to system objects, i.e. SWITCHes , RECEIVERs or MOTORs, in three ways:

(1) it belongs to a class and can respond to commands prescribed for that class. For example, systems objects RECEIVER 1, ..., RECEIVER 8 are all instances of class RECEIVERS and each RECEIVER can respond to the set of commands prescribed for the class RECEIVERS, which are STATE, KEEPCOUNT, COUNT and CLEARCOUNT.

(2) it can have its own variables, which are accessible only by the object itself. For example, each RECEIVER and MOTOR has

its own variable COUNT and its value can be accessed or changed only by sending the object a command.

(3) the same syntax is used for sending commands to user defined objects as is used for system objects,

i.e. <object name> ! <command for object>

User defined objects have special features that are very important for multi-programming, which will be described later on in this chapter.

4.4.1 Example: DC Motor

This example shows why objects are desirable and how a new class of objects, DC motors, may be created.

A DC motor can be conveniently operated using two SWITCHes. One SWITCH (power switch) is for switching a motor on and off and another SWITCH (direction switch) is for changing the motor's direction of rotation. Figure 4.1 shows the relationship between the states of the SWITCHes and the states of the DC motor.

direction switch	power switch	MOTOR
On	On	turn clockwise
Off	On	turn anticlockwise
On	Off	stop
Off	Off	stop

Figure 4.1 Controlling DC motor

For example, if SWITCH 1 is used as a direction switch and SWITCH 2 is used as a power switch, the command

```
SWITCH 1! ON; SWITCH 2! ON
```

would make the motor turn clockwise, and the command

```
SWITCH 1! OFF; SWITCH 2! ON
```

would make the motor turn anticlockwise. If these commands are to be used often it would be better to encapsulate them in procedures called ANTICLOCKWISE, CLOCKWISE and HALT, or something similar.

However, if there are a number of DC motors to be controlled the procedures have to be extended to input values which specify the SWITCHes' numbers. The CLOCKWISE procedures could be defined as

```
CLOCKWISE 'DIRECTION 'POWER;
SWITCH :POWER ! ON; SWITCH :DIRECTION ! OFF
```

Assuming that there are four motors, the odd numbered SWITCHes are used as direction switches and the even numbered SWITCHes as power switches, then the command

```
CLOCKWISE 1 2
```

would make the first motor turn clockwise, and the command

```
CLOCKWISE 3 4
```

would make the second motor turn clockwise. It would be better to create a new class of DC motor objects so that each DC motor remembers which are its direction and power switches and can respond to appropriate commands.

The new class to be created is called DC-MOTOR:

```
NEWCLASS 'DC-MOTOR HAS 'DIRECTION 'POWER 'STATE
```

When an object of this class is created it will be associated with three variables: DIRECTION, POWER, and STATE, which are accessible only by the object itself. The variables DIRECTION and POWER are for storing the numbers for direction and power switches respectively. The variable STATE is for storing information about what the motor is currently doing, which could be stationary, turning clockwise or turning anticlockwise.

Every object of this class can respond to five commands: READY, TURNC, TURNA, STOP and STATE. READY is to initialise a motor; TURNC is to make a motor turn clockwise; TURNA is to make a motor turn anticlockwise; STOP is to stop a motor turning and STATE returns the state of a motor. Notice that commands associated with the class DC-MOTOR can share the same names with the class stepping MOTOR.

The commands can be defined as

```
DEFINE 'READY CLASS 'DC-MOTOR

READY 'X 'Y;
MAKE 'DIRECTION :X;
MAKE 'POWER :Y;
MAKE 'STATE [STATIONARY]

DEFINE 'TURNC CLASS 'DC-MOTOR

TURNC;
SWITCH :DIRECTION! ON;
SWITCH :POWER! ON;
MAKE 'STATE [TURNING CLOCKWISE]

DEFINE 'TURNA CLASS 'DC-MOTOR

TURNA;
SWITCH :DIRECTION! OFF;
SWITCH :POWER! ON;
MAKE 'STATE [TURNING ANTICLOCKWISE]

DEFINE 'STOP CLASS 'DC-MOTOR

STOP;
SWITCH :POWER! OFF;
MAKE 'STATE [STATIONARY]

DEFINE 'STATE CLASS 'DC-MOTOR

STATE;
RETURN :STATE
```

Listing 4.1 DC motor programs

Note, one limitation in the prototype implementation is that user defined objects must have different names. They cannot be distinguished using subscripts, like the system objects. Four DC-MOTOR objects would be creted as follows:

```
NEWOBJECT 'MOTOR-1 CLASS 'DC-MOTOR
NEWOBJECT 'MOTOR-2 CLASS 'DC-MOTOR
NEWOBJECT 'MOTOR-3 CLASS 'DC-MOTOR
NEWOBJECT 'MOTOR-4 CLASS 'DC-MOTOR
```

Listing 4.2 shows an example of communicating with these objects. The computer's prompt is 'W:', it stands for 'Waiting:' and the user's input is in boldface.

```
    @initialise the motors

    @MOTOR-1's direction and power switches are SWITCHes
    @1 and 2 respectively
W: MOTOR-1! READY 1 2

    @MOTOR-2's direction and power switches are SWITCHes
    @3 and 4 respectively
W: MOTOR-2! READY 3 4

    @MOTOR-3's direction and power switches are SWITCHes
    @5 and 6 respectively
W: MOTOR-3! READY 5 6

    @MOTOR-4's direction and power switches are SWITCHes
    @7 and 8 respectively
W: MOTOR-4! READY 7 8

    @make MOTOR-1 turn clockwise
W: MOTOR-1! TURNC

    @make MOTOR-2 turn anticlockwise
W: MOTOR-2! TURNA


W: PRINT MOTOR-1! STATE
   TURNING CLOCKWISE

W: PRINT MOTOR-2! STATE
   TURNING ANTICLOCKWISE

W: PRINT MOTOR-3! STATE
   STATIONARY

    @make MOTOR-1 stop turning
W: MOTOR-1! STOP

W: PRINT MOTOR-1! STATE
   STATIONARY
```

Listing 4.2 Manipulating DC motors

4.5 Multi-programming

In Concurrent-Logo the parallel bar notation '//' is used to initiate processes that are to be executed in parallel. For example, the command

```
REPEAT 10 (PRINT 'BURGLAR) // REPEAT 10 (SOUND)
```

would print the word BURGLAR on the screen and beep at the same time. A parallel statement is completed when all the processes that it initiated have been terminated. Consider the command

```
REPEAT 1000 (PRINT 'WAIT) // REPEAT 2 (SOUND); PRINT 'FINISHED
```

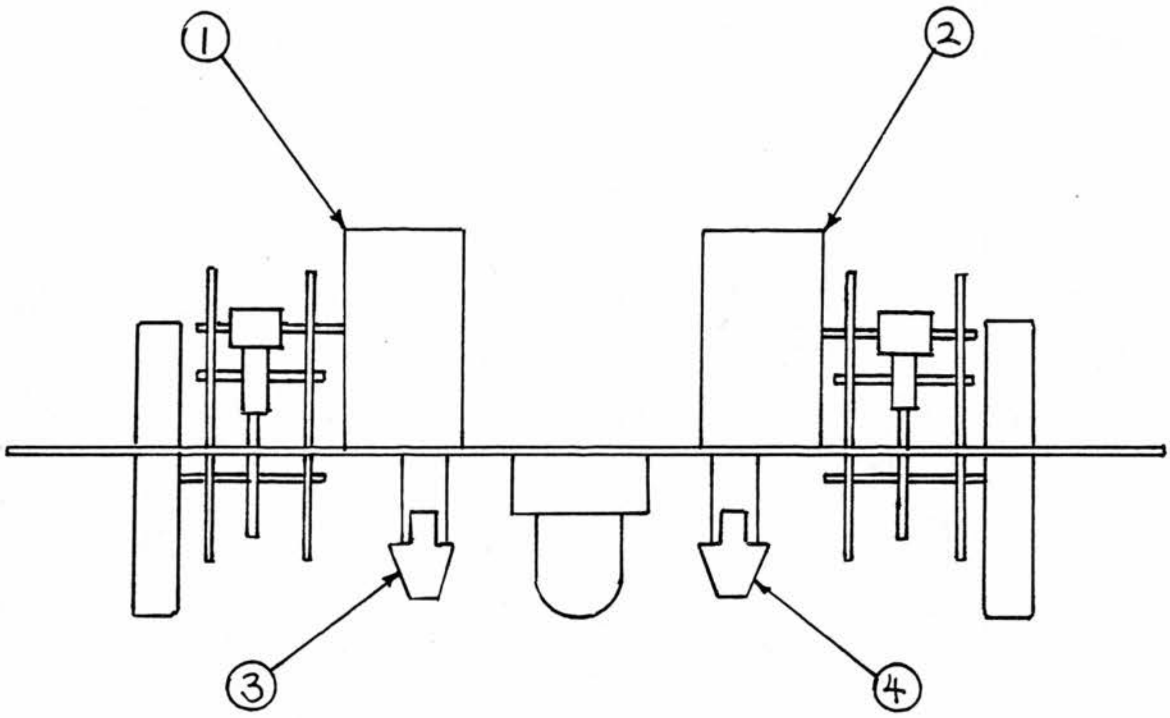
which is semantically equivalent to

```
(REPEAT 1000 (PRINT 'WAIT) // REPEAT 2 (SOUND)); PRINT 'FINISHED
```

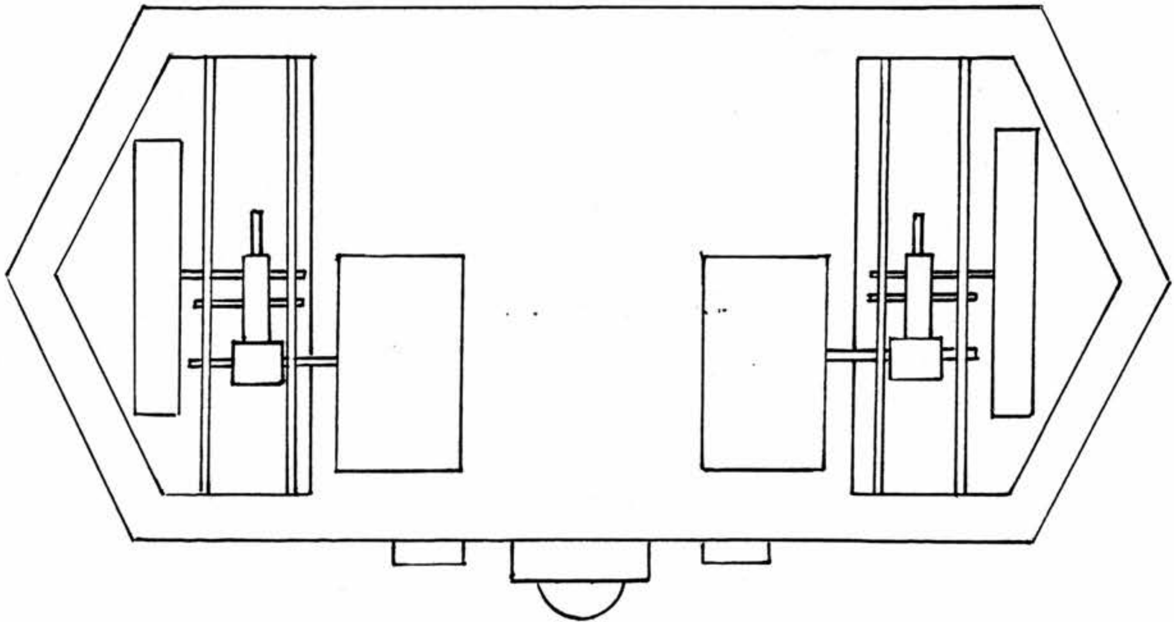
would print the word FINISHED only after one thousand WAITs had been printed and not immediately after two beeps had been made. The prototype implementation allows up to eight processes to run in parallel.

4.5.1. Example: controlling a turtle

A turtle (see figure 4.2) can be built out of Meccano. It has two stepping motors mounted, back to back, on it. The left motor is controlled by MOTOR 1 and right motor is controlled by MOTOR 2. Since the motors are mounted back to back, the turtle moves in a straight line when they are rotating in opposite directions; it turns on the spot when they are rotating in the same direction.



Front View



Top View

Figure 4.2 Turtle

Component	Part No.
Motor	1 and 2
Reflective-opto switch	3 and 4

Scale 1:2.5 (approximate)

Note: The Turtles referred to in sections 4.5.1 and 6.4 did not have parts 3 and 4 mounted on them.

The procedures FORWARD, BACKWARD, LEFT and RIGHT (listing 4.3) are defined to make the turtle move forward, backward, left and right respectively. The unit of the distance moved and the unit of the amount turned is arbitrary.

```
FORWARD 'X;
MOTOR 1! TURNC :X // MOTOR 2! TURNA :X

BACKWARD 'X;
MOTOR 1! TURNA :X // MOTOR 2! TURNC :X

LEFT 'X;
MOTOR 1! TURNA :X // MOTOR 2! TURNA :X

RIGHT 'X;
MOTOR 1! TURNC :X // MOTOR 2! TURNC :X
```

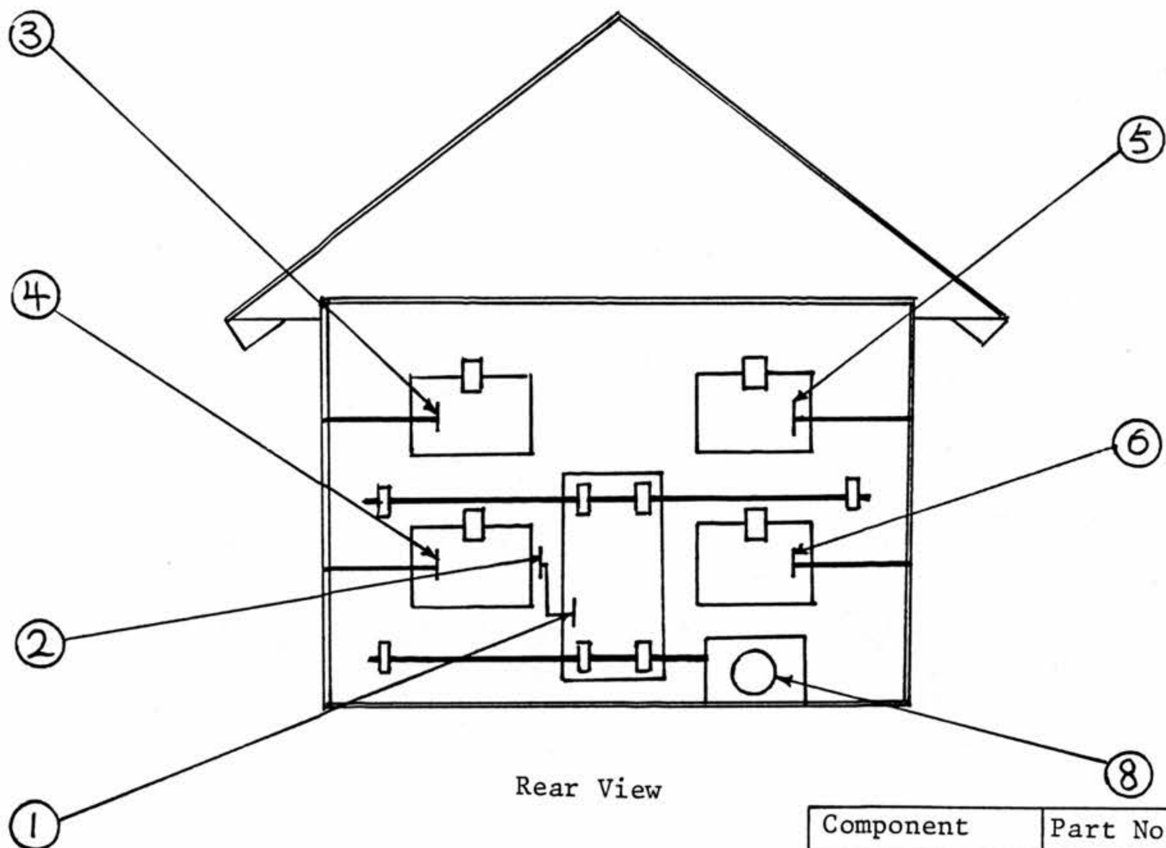
Listing 4.3 Turtle programs

4.5.2. Example: a security system

A doll's house (figure 4.3) can be built out of Meccano. It has one DC motor for opening and closing the sliding door; two reed switches for detecting the closed and open positions of the door; four micro-switches, one behind each window; one button-switch, used as a door bell.

In this example:

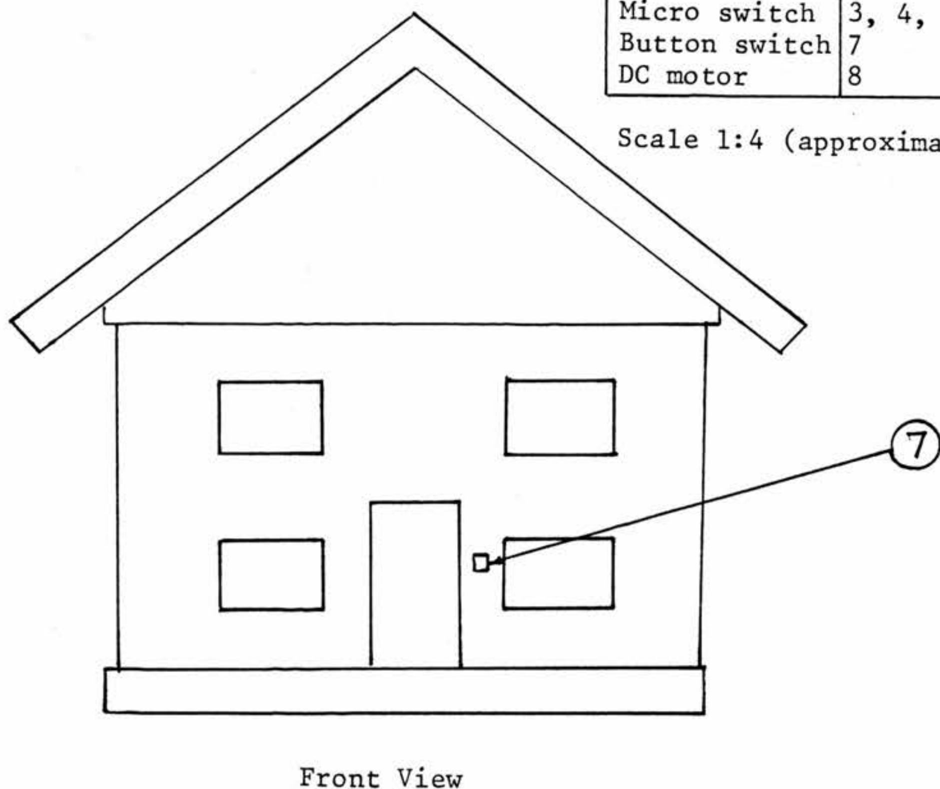
SWITCH 1 is for turning the DC motor on and off;
SWITCH 2 is for controlling the motor's direction of rotation;
RECEIVERs 1 and 2 are connected to the reed switches;
RECEIVERs 3 to 6 are connected to the micro-switches;
RECEIVER 7 is connected to the door bell.



Rear View

Component	Part No.
Reed switch	1 and 2
Micro switch	3, 4, 5 and 6
Button switch	7
DC motor	8

Scale 1:4 (approximate)



Front View

Figure 4.3 Doll's house

The procedure WINDOW (listing 4.4) would make the computer detect a burglar trying to break in through any of the windows. Once a window is open the computer would print out a message and beep continuously.

```
WINDOW;
WHENEVER EQU? RECEIVER 3! STATE 'ON
 (PRINT [BREAK IN AT THE TOP RIGHT WINDOW];    FOREVER (SOUND))//
WHENEVER EQU? RECEIVER 4! STATE 'ON
 (PRINT [BREAK IN AT THE BOTTOM RIGHT WINDOW]; FOREVER (SOUND))//
WHENEVER EQU? RECEIVER 5! STATE 'ON
 (PRINT [BREAK IN AT THE TOP LEFT WINDOW];     FOREVER (SOUND))//
WHENEVER EQU? RECEIVER 6! STATE 'ON
 (PRINT [BREAK IN AT THE BOTTOM LEFT WINDOW];  FOREVER (SOUND))
```

Listing 4.4 WINDOW program

The computer could also be programmed to be a door keeper. The function of DOOR and its sub-procedures (listing 4.5) is to ask for a secret word or sentence whenever the door bell is pressed. If the correct answer is typed then the door slides open, otherwise the door remains closed.

```
DOOR;
@whenever doorbell is pressed ask for password
WHENEVER EQU? 'ON RECEIVER 7! STATE (PASSWORD)

PASSWORD;
MAKE 'X ASK [WHAT IS THE PASSWORD?];
IF EQU? :X [GOOD] (OPENDOOR; CLOSEDOOR)
              ELSE (PRINT [TRY AGAIN PLEASE])

OPENDOOR;
SWITCH 2! ON;
SWITCH 1! ONUNTIL EQU? RECEIVER 2! STATE 'ON;

CLOSEDOOR;
SWITCH 2! OFF;
SWITCH 1! ONUNTIL EQU? RECEIVER 1! STATE 'ON;
```

Listing 4.5 DOOR program

The alarm and automatic door system can be simply combined into one program:

```
HOUSE;
DOOR // WINDOW
```

Listing 4.6 HOUSE program

4.5.3. Problems with multi-programming

In the previous examples, the processes that executed concurrently were independent, i.e. they did not exchange information and the order of events was independent. However, when concurrent processes do depend on one another they pose some problems. This section describes the problems briefly, and the following section explains the facilities provided in Concurrent-Logo for handling them. For detailed discussion on the problems of parallel processing see Ben-Ari (1982).

The first problem is called 'mutual exclusion'. When concurrent processes share one or more variables there is a danger that they may update the same variable simultaneously. When this happens data would be corrupted and lead to error.

The following simple example demonstrates the problem:

```
MAKE 'X [];
REPEAT 10 (MAKE 'X PUTF 'A :X) // REPEAT 10 (MAKE 'X PUTF 'B :X)
```

After the command is executed, the value of the shared variable 'X is a list of only 10 elements rather then the expected value, a list of 20 elements. Since both concurrent processes tried to expand the list at the same time they just overwrote each other and some data were lost. Therefore, it is necessary to provide a facility called 'mutual exclusion', to ensure secure access to shared variables.

The other problem associated with multi-programming is 'synchronization'. When concurrent processes are running, there are occasions when a process cannot continue until a certain event has taken place. Therefore, it is necessary for a process to signal either that it is waiting on an event, or that the event has occurred so that other processes which are waiting may continue.

A well known example that requires both mutual exclusion and synchronization is the **Producer-Consumer problem** (Dijkstra, 1968) with a bounded buffer. The bounded buffer is a data structure that can hold only a finite number of elements. Two cyclic processes running in parallel access the buffer. The first of these is the producer: it produces a new element and appends it to the sequence produced so far. The other is the consumer: it removes the first element from the sequence. Mutual exclusion is required when either process accesses the buffer. Synchronization is also required. If the buffer is full, the producer has to wait for the consumer to remove an item; if the buffer is empty, the consumer has to wait for the producer to insert an item. In practice, there may be more than one consumer and producer. Therefore, a queue of processes is associated with the wait condition.

4.5.4. Objects revisited

The design of the facilities for cooperating sequential processes is a modification of Hoare's (1974) **monitor** concept. A monitor defines a shared data structure and a set of procedures that can operate on it. Processes cannot manipulate the shared data structure directly, but have to call the monitor procedures. If more than one procedure calls the monitor procedures simultaneously then

these procedure calls will be executed strictly one at a time to avoid data corruption. Therefore a monitor is like an object with scheduling facilities.

In Concurrent-Logo the object model is extended to deal with the problems in multi-programming. The mutual-exclusion problem stated in terms of the object model would be 'If several commands were sent to an object simultaneously, how would the object respond?' In this situation a simple solution would be for the object to obey the commands one at a time. Though simple, the solution is reasonable because people do this naturally. For example, if two persons were to speak to me simultaneously, I would get hopelessly confused and would say to them, 'Please! one at a time.' The synchronization problem stated in terms of the object model would be, 'After an object had accepted a command and had also realized that it could not carry out the required task, what would the object do?' If the object cannot fulfil a request due to some condition, the request will be delayed, and the object will make itself available to other requests. After a request has been successfully processed the object will check whether it can restart any previously delayed requests.

There are certain features in this implementation of objects which help to overcome the problems:

(1) an automatic scheduling mechanism - if an object receives more than one message simultaneously, they will be processed strictly one at a time

(2) the DELAYIF control statement:

Syntax: DELAYIF <condition>

Action: while obeying a message, if the object encounters the DELAYIF statement and the condition is 'TRUE, the request is delayed, and the object makes itself available to other requests. After a request has been successfully processed the object will check whether it can restart any previously delayed requests.

To solve the Producer-Consumer problem, all that is required is an object, say of class BUFFER-HANDLER, which knows how to insert an element into or remove an element from the buffer.

Let the maximum size of the buffer be 20. The class BUFFER-HANDLER and the INSERT and REMOVE messages may be created and defined as follows:

Creating a new class BUFFER-HANDLER:

```
NEWCLASS 'BUFFER-HANDLER HAS 'BUFFER


DEFINE 'INSERT CLASS 'BUFFER-HANDLER

INSERT 'ELEMENT;
DELAYIF EQU? LENGTH :BUFFER 20;
MAKE 'BUFFER PUTLAST :ELEMENT :BUFFER
```

The insertion of an element will be delayed if the buffer is full, otherwise the element will be inserted at the end of the BUFFER.

```
DEFINE 'REMOVE CLASS 'BUFFER-HANDLER

REMOVE;
DELAYIF EMP? :BUFFER;
LOCAL 'TEMP;
MAKE 'TEMP FIRST :BUFFER;
MAKE 'BUFFER REST :BUFFER;
RESULT :TEMP
```

Listing 4.7 Class: BUFFER-HANDLER

If the BUFFER is empty the message will be delayed. Otherwise the value of the first element in the BUFFER is returned.

The producers and consumers, instead of having bits of code to manipulate the buffer and bits of code to ensure mutual exclusion and synchronization, need only send messages to an object of class BUFFER-HANDLER requesting insertion or removal of data.

A BUFFER-HANDLER object SCHEDULER can be created by the command

```
NEWOBJECT 'SCHEDULER CLASS 'BUFFER-HANDLER.
```

An example:

```
REPEAT 10 (SCHEDULER! INSERT 'A) //
                        REPEAT 10 (SCHEDULER! INSERT 'B)
```

4.5.5. Example: Controlling a lift

To demonstrate the more elaborate use of multi-programming, this example considers writing a program for controlling a model lift (figure 4.4) which can be built out of Meccano. It has one DC motor and three reed switches fitted to it. The motor is used to drive the lift cage up and down. The reed switches detect whether the lift cage is at a particular floor. Three button-switches control the lift. The idea is to have the computer detect signals from the three button switches so that whenever

(1) the left switch is pressed the lift cage moves to the ground floor.

(2) the middle switch is pressed the lift cage moves to the second floor.

(3) the right switch is pressed the lift cage moves to the top floor.

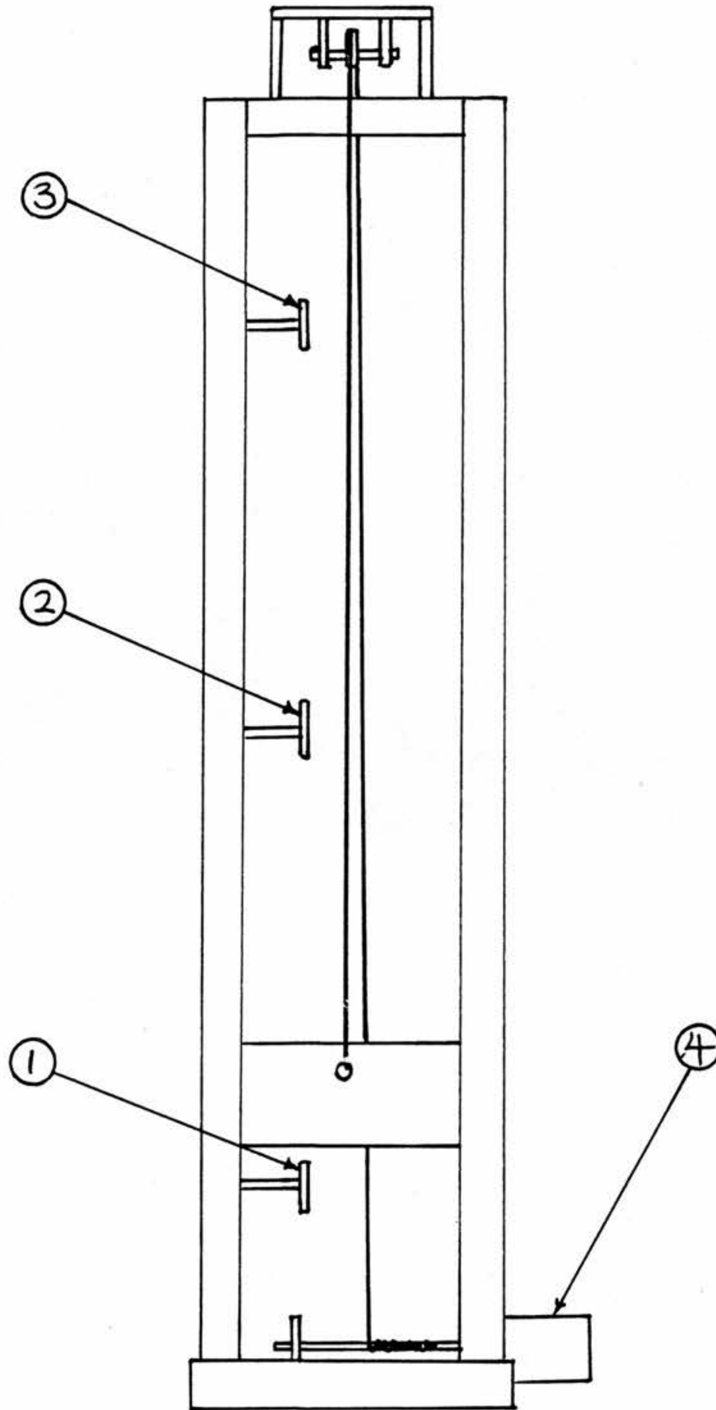


Figure 4.4 Lift

Component	Part No.
Reed switch DC motor	1,2 and 3 4

Scale 1:4 (approximate)

The function of the required program seems rather simple: detect the occurrence of a signal, then respond to it by actuating the motor so that the lift cage is moved to the appropriate floor. However, an underlying question is: What should the response be if signals occur while the lift cage is moving? A trivial but unsatisfactory solution is to ignore all signals while the computer is busy controlling the lift cage. Another solution is to keep a record of all signals according to some scheduling procedures and make the lift cage move from floor to floor accordingly. The complexity of the final program depends very much on the control structures and the scheduling algorithm used.

Opting for the second solution, an obvious but cumbersome way to write the program is to treat it as a single process and use many nested conditional and iterative statements. The resulting program might be difficult to understand and debug. It is more elegant to make use of multi-programming. The program is decomposed into three main components: a signal detector, a scheduler and a lift controller. The way that these components interact is shown in figure 4.5.

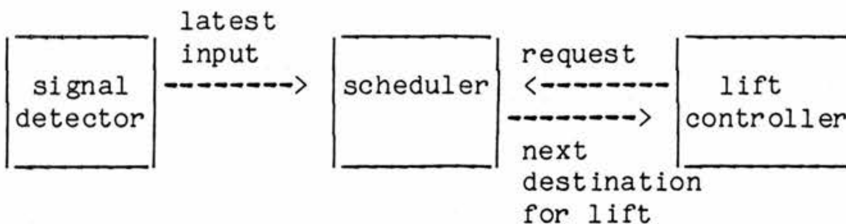


Figure 4.5 A multi-programming solution to the lift problem

The signal detector and lift controller are two parallel processes.

The signal detector continuously checks if any of the switches is pressed. Once a signal has been detected it passes the relevant information to the scheduler. The lift controller is dedicated to controlling the lift cage. Initially the lift controller sends a request to the scheduler and the scheduler replies with the floor number that the lift cage should move to. On receiving a reply the lift controller starts the lift cage moving accordingly. Once the lift cage has stopped at its destination the next request is sent and the same process continues cyclically. It is apparent that this is similar to the Producer-Consumer problem discussed in section 4.5.3 - the signal detector being the producer and the lift controller being the consumer. To simplify this example, the object SCHEDULER, of class BUFFER-HANDLER, as defined in section 4.5.4 is used.

Let the electrical and electronic components be connected as follows:

(1) connect the reed switch at the bottom floor to RECEIVER 1.

(2) connect the reed switch at the second floor to RECEIVER 2.

(3) connect the reed switch at the top floor to RECEIVER 3.

(4) connect the bottom floor button switch to RECEIVER 4.

(5) connect the second floor button switch to RECEIVER 5.

(6) connect the top floor button switch to RECEIVER 6.

(7) use SWITCH 1 for turning the motor on and off.

(8) use SWITCH 2 for controlling the motor's direction of rotation.

The definition of the SIGNAL-DETECTOR is:

```
SIGNAL-DETECTOR;
FOREVER
  (IF EQU? 'ON RECEIVER 4! STATE (SCHEDULER! INSERT 1);
   IF EQU? 'ON RECEIVER 5! STATE (SCHEDULER! INSERT 2);
   IF EQU? 'ON RECEIVER 6! STATE (SCHEDULER! INSERT 3))
```

The definition of the LIFT-CONTROLLER and its sub-procedures are as follows:

```
LIFT-CONTROLLER;
LOCAL 'CURRENT;
LIFT-READY;
LIFT-CONTROL

LIFT-READY;
    @to initialise, move the lift to the ground floor
    @and set the variable 'CURRENT to 1}
SWITCH 2! OFF;
SWITCH 1! ONUNTIL EQU? 'ON RECEIVER 1! STATE;
MAKE 'CURRENT 1

LIFT-CONTROL;
LOCAL 'NEXT;
FOREVER
  (MAKE 'NEXT SCHEDULER! REMOVE;
   IF NOT EQU? :NEXT :CURRENT
     (IF GRE? :NEXT :CURRENT (SWITCH 2! ON) ELSE (SWITCH 2! OFF);
      SWITCH 1! ONUNTIL EQU? 'ON RECEIVER :NEXT ! STATE;
      MAKE 'CURRENT :NEXT))
```

Listing 4.9 LIFT program

To start the program, the user types

```
SIGNAL-DETECTOR // LIFT-CONTROLLER
```

This example illustrates how the use of multi-programming facilities greatly simplifies and imposes structures on the program.

4.6 Event handling

In computing terms, a demon may be defined as a 'module that is automatically activated when a certain condition becomes true' (Bobrow and Raphael, 1974). The Concurrent-Logo implementation of this concept is called GUARD. The word 'guard' is chosen because it is much friendlier than 'demon'. It also has the implication of watching out for an event to happen. A guard is created when it is told to remember a set of commands. Once it is created it responds

to two messages WAKEUP and SLEEP. It is in a passive state until it is told to wake up. When a guard is awake, it obeys the set of commands it was previously told to remember. It returns to the passive state either when it finishes obeying the commands or it receives a message telling it to sleep.

4.6.1 Example: Digital clock

This example shows how a guard called CLOCK is created. When CLOCK is awake, it updates a digital clock display, at the top right hand corner of the screen, once a second.

```
@Enter the editor to define a set of commands for CLOCK
TELL CLOCK

CLOCK;
@Initialise the clock
INITIALIZE;
@Update the clock every second
WHENEVER NEXT.SECOND (UPDATE)
```

Listing 4.10 Guard: CLOCK

The guard facility makes it very easy to run another procedure while the clock is ticking. For example, to run the clock and a procedure called HANGMAN the command is simply

```
CLOCK ! WAKEUP ; HANGMAN
```

If required, the CLOCK can be switched on and off many times just by sending it commands WAKEUP or SLEEP accordingly. The control flow of the HANGMAN procedure would be very obscure if the clock was to be built into it without using the guard facility.

4.6.2 The lift problem revisited

This example shows how the solution to the lift problem as described in section 4.5.3 can be made more efficient using guards.

The solution, as it was, would detect a key-press and append the corresponding floor number at the end of the list 'BUFFER. The program permits BUFFER to have a list which has multiple instances of the same number, for example [1 3 1 1 3 1 2]. However, it is unnecessary, or even wrong. The above list could be reduced to [1 3 2] since, no matter how many times a request is made, the lift only has to move to the requested floor once and all the occurrences of the same request are satisfied. The problem can be overcome by modifying the SCHEDULER's INSERT routine so that it checks whether an input value is already a member of the list.

A more elegant solution is based on the observation that once a signal is detected and the corresponding floor number is inserted into 'BUFFER it is not necessary to detect further occurrences of the same signal until the request has been satisfied. As a result, the value of 'BUFFER will never have more than three elements.

Instead of having one signal detecting routine, we can have three guards. Each is responsible for the detection of a particular signal. A guard will stay awake until it has received a signal. It will be woken up again only after the lift has responded to the signal. So the program would be as follows:

```
TELL DETECT-1
WHILE EQU? 'OFF RECEIVER 4! STATE ();
SCHEDULER! INSERT 1

TELL DETECT-2
```

```
WHILE EQU? 'OFF RECEIVER 5! STATE ();
SCHEDULER! INSERT 2

TELL DETECT-3
WHILE EQU? 'OFF RECEIVER 6! STATE ();
SCHEDULER! INSERT 3
```

The new definitions of LIFT-CONTROLLER and its sub-procedures are:

```
LIFT-CONTROLLER;
LOCAL 'NEXT 'CURRENT;
LIFT-READY;
LIFT-CONTROL

LIFT-READY;
@to initialize, move the lift to the ground floor
@and set the variable 'current to 1
SWITCH 2! OFF;
SWITCH 1! ONUNTIL EQU? 'ON RECEIVER 1! STATE;
MAKE 'CURRENT 1;
DETECT-2! WAKEUP;
DETECT-3! WAKEUP

LIFT-CONTROL;
FOREVER
   (MAKE 'NEXT SCHEDULER! REMOVE;
    IF EQU? :CURRENT 1 (DETECT-1! WAKEUP);
    IF EQU? :CURRENT 2 (DETECT-2! WAKEUP);
    IF EQU? :CURRENT 3 (DETECT-3! WAKEUP);
    IF GRE? :NEXT :CURRENT (SWITCH 2! ON) ELSE (SWITCH 2! OFF);
    SWITCH 1! ONUNTIL EQU? 'ON RECEIVER :NEXT! STATE;
    MAKE 'CURRENT :NEXT)
```

Listing 4.11 LIFT program using guards

To start the program, the user now types:

```
LIFT-CONTROLLER
```

4.7 Related work

Concurrent-Logo is already showing its influence and has found commercial expression. There are now two versions of Logo, Control-Logo and Nimbus Logo, which provide some of the facilities found in Concurrent-Logo.

Control-Logo has been developed by the Advisory Unit For Computer Based Education (AUCBE) and the work was started during the final implementation phase of Concurrent-Logo. The workers at AUCBE share with the author the view that Logo is a good computer language that can be suitably extended for learning control applications. However, instead of designing and implementing a new version of Logo they took a short cut. They created Control-Logo by implementing machine code utilities and linking them to existing full implementations of Logo. These include RML Logo and Spectrum Logo. The new Logo commands for invoking the machine code are:

SENDPORT	outputs a byte
READPORT	inputs a byte
TURNON	switches on a bit
TURNOFF	switches off a bit
STATE	checks the state of a bit
COUNT	counts pulses on specified input bit.

Due to the limitations of the original Logo implementations, Control-Logo provides neither multi-programming facilities nor commands for controlling stepping motors. As a result, Control-Logo is rather limited.

Nimbus Logo has been developed at Edinburgh University for the

Nimbus personal computer[10], which is a 16-bit machine. Concurrent-Logo has had a direct influence on the design of Nimbus Logo, in particular the ideas of introducing more powerful control structures and multi-programming into Logo. Nimbus Logo has sprites but does not allow users to create any other types of objects because Nimbus Logo is designed as an enhancement of RML Logo, and was subject to the constraint that it should not differ too much in fundamental design and should be compatible with the earlier system.

Nimbus Logo incorporates directly the FOREVER and WHENEVER commands from Concurrent-Logo and introduces a new AWAIT command. Its syntax is

```
AWAIT <condition>
```

which holds up the process until the condition is true. Multiple processes can be started using the PARALLEL command. For example, the commands

```
PRINT 'START
PARALLEL [ [REPEAT 10 [PRINT 'A]] [REPEAT 20 [PRINT 'B]] ]
PRINT 'FINISH
```

would print the word START then print 10 As and 20 Bs simultaneously. When all the As and Bs had been printed the word FINISH would be printed finally. The above commands are equivalent to the Concurrent-Logo commands:

```
PRINT 'START;
REPEAT 10 (PRINT 'A) // REPEAT 20 (PRINT 'B);
PRINT 'FINISH
```

---

[10] Nimbus personal computer is manufactured by Research Machines Ltd. (RML).

Instead of GUARDs, Nimbus Logo provides a BEGIN command. Its syntax is

```
BEGIN [action]
```

which starts a process executing 'action' in parallel with the existing process. For example, the commands

```
BEGIN [CLOCK] HANGMAN
```

would start the CLOCK procedure, which might be defined to update a clock display on the screen once a second, running in the background and then carry on executing the HANGMAN procedure. These are similar to the Concurrent-Logo commands

```
CLOCK ! WAKEUP; HANGMAN
```

The difference between BEGIN and GUARDs is that a process started by a BEGIN command will stop running only when it reaches the logical end; a GUARD terminates when it reaches the logical end or when it recives the SLEEP command. For example, there is no direct equivalent in Nimbus Logo of

```
@initalise the clock then start game 1
CLOCK ! WAKEUP; GAME_1; CLOCK ! SLEEP;
@stop the clock then reinitialise it and start game 2
CLOCK ! WAKEUP; GAME_2; CLOCK ! SLEEP
```

Nimbus Logo provides facilities for linking user defined machine code routines. The commands are:

```
BLOAD          @load a machine code file
UNBLOAD        @delete a machine code file from core
DRIVER         @load a turtle driver
NODRIVER       @remove the turtle driver from core
```

The machine code files have to be in a special format which is fully described in the documentation supplied with Nimbus Logo. Therefore, extending the I/O commands for control applications should be simple. Since Nimbus Logo has also been designed for business use, it provides very good file handling facilities.

CHAPTER 5

A PILOT STUDY

5.1 Aims

After the prototype of Concurrent-Logo had been implemented, it was used in a pilot study to teach control applications.

The study had three aims:

(1) to develop ideas for a course in control applications. To carry out the study the author had developed a series of project ideas (see section 5.5.2). This experimental course might form the basis for curriculum developers to modify or extend in the future.

(2) to identify the advantages and difficulties of learning control applications through programming.

(3) to evaluate Concurrent-Logo. Designing a programming language, though guided by principles, is a very subjective process. Therefore, the best way to assess Concurrent-Logo was to put it to practical use.

The goal that united these aims was to develop a practical classroom system for teaching control applications.

5.2 Design

The prototype of Concurrent-Logo was implemented on a TERAK 8510 microcomputer. It was chosen for several reasons. First, it is very reliable. Second, it is a sixteen bit machine with 28K word of random access memory, which is more powerful than the popular eight bit microcomputers. At the time of implementation it served as good

interim hardware for developing programming system for the next generation of sixteen bit microcomputers for schools. Third, several TERAKs were readily available to the author.

Lothian Region Education Authority gave the author permission to carry out a small scale evaluation study in a local secondary school - Firrhill High School. The study began on 24th October 1983 and extended through to 29th May 1984. Two groups of students were selected from the school. One group of five students was drawn from the fourth year (approximately fifteen years of age) and the other group of seven students was taken from the third year (approximately fourteen years of age).

During the study, two TERAK microcomputers and the necessary hardware were put in the school.

The course consisted of a collection of six projects. Each project involved the students in writing programs for a particular control device. The control devices were: windmill, turtle, dolls' house, lift, turtle with optical sensors and robot arm.

Every week, except during holidays and examinations, each group had a seventy-five-minute session with the author. The fourth year group had seventeen sessions; the third year group had twenty four sessions. The former group had fewer sessions because they had to spend more time preparing for their examinations. The particpants had no access to the TERAKs other than during the allocated time. However, some students had their own computers or belonged to the school's computer club so they would have done some computing in between the sessions.

During each session, the group was divided into three sub-groups so that at any time there were two sub-groups working on the computer and one waiting. The sub-grouping remained the same from the third session onwards.

The classroom tutoring was done entirely by the author. Worksheets (Appendix I) were produced to introduce the students to new programming concepts, new control devices and project ideas. No formal lectures or discussion groups were held.

From the point of experimental design it would have seemed better if the tutoring had been done by an independent teacher rather than by the author. The teacher would have been able to provide impartial comments, with the author concentrating on observing and recording information. However, this was not possible because time and finance did not allow recruitment and training of a teacher. From time to time, the head physics teacher, who helped to set up the experiment, came into the classroom to observe, and outside the teaching time he also gathered comments from the students through informal conversations.

Evaluation of the study was based on classroom observation, record of students' work, questionnaires and a post-test.

5.3 Participants

Initially two groups of six students were selected: one from the fourth year and the other from the third year. After the first week, because of ill health, a boy dropped out of the fourth year group, and a girl was added to the third year group. Therefore the fourth and third year group had five and seven students respectively.

Small experimental groups were used to enable the author to monitor the progress of each student. The selection was done independently by the school's Principal Teacher of physics, and no attempt was made to influence his decision. The criterion for selection was that the particpants should represent a spread of learning ability over both groups.

All of the students expressed interest in knowing more about computing. Each student's computing experience prior to the course and his/her learning ability is shown in figures 5.1 and 5.2. The information concerning the students' computing experience was obtained from the use of a questionnaire (Questionnaire I, Appendix II), and from the principal teacher's comments. Each student's learning ability is graded by his/her own teacher. The grading scale is:

a) well above average
b) above average
c) just above average
d) average
e) just below average
f) below average
g) well below average

NAME	COMPUTING EXPERIENCE	LEARNING ABILITY
Nigel	Very experienced in BASIC programming.	b
Neil	He has played computer games and copied BASIC programs from books.	b
Willie	He has been using computers for six years and has a special interest in computer hardware. On his own initiative, he took an 'O' grade in electronics. He also has experience in using BASIC and Pascal.	c/d
Martin	He has written simple programs in BASIC.	d
Heath	He has no computing experience at all.	e

Figure 5.1 Fourth year group

NAME	COMPUTING EXPERIENCE	LEARNING ABILITY
Michael	He has played computer games.	a
Keith	He has written simple programs in BASIC.	a
Kevin	He has written simple programs in BASIC.	b
Lynette	She has written simple programs in BASIC.	b
Gary	He has written simple programs in BASIC.	d
Heather	She has played computer games.	d
Ruth	She has written simple programs in BASIC.	e

Figure 5.2 Third year group

5.4 Equipment

5.4.1 Computer

Two TERAK 8510 microcomputers were used in the study. Each computer had a single density eight inch disk drive and was extended with a parallel I/O board[11].

[11] The parallel I/O board is manufactured by Grant Technology Systems Corporation, U.S.A.

5.4.2 Control Device

Six control devices were used in the study. The devices were: windmill, turtle, doll's house, lift, turtle with optical sensors and robot arm. All the devices, except the robot arm, were designed and built by the author using Meccano. The robot arm is Armdroid.

The electronic components that were used for the Meccano devices were DC motors, button-switches, reed switches, micro-switches, reflective opto-switches and stepping motors[12].

Windmill

The windmill (see figure 2.2 in chapter 2) had one DC motor for making the sails spin.

Turtle I

The turtle (see figure 4.2 in chapter 4) had two DC motors mounted on it. The motor on the right hand side was used to drive the right wheel, and the motor on the left hand side drove the left wheel. Notice the motors were mounted back to back, i.e. when they turned in opposite directions the turtle moved in a straight line; when they turned in the same direction the turtle turned on the spot.

Lift

The lift (see figure 4.4 in chapter 4) had one DC motor and three reed switches mounted on it. The motor was used to drive the

[12] All components, except the stepping motors, were obtained from Radiospares. The stepping motors were four-phase 12 volt motors manufactured by Philips.

lift cage up and down. Reed switches detected whether the lift cage was at a particular location.

Doll's house

The doll's house (see figure 4.3 in chapter 4) had four micro-switches, one behind each window, so that whenever a window was pushed open a corresponding micro-switch was triggered; one button-switch was used as a door bell; one DC motor opened and closed the sliding door; two reed switches detected the closed and open positions of the door.

Turtle II

The advanced turtle (see figure 4.2 in chapter 4) had two stepping motors, instead of DC motors, mounted on it. Stepping motors were used to give the students some control over how much the motors should turn. Two reflective-opto switches were fixed to its front. A reflective-opto switch sends an ON signal when it is above a black surface and it sends an OFF signal when it is above a white surface.

Robot arm

Armdroid (figure 5.3) had five moving parts: fingers, wrist, forearm, upper arm and shoulder, controlled by six stepping motors. Each moving part, except the wrist, was controlled by one motor. The wrist was controlled by two motors. The same kind of stepping motor was used for Turtle II and the arm.

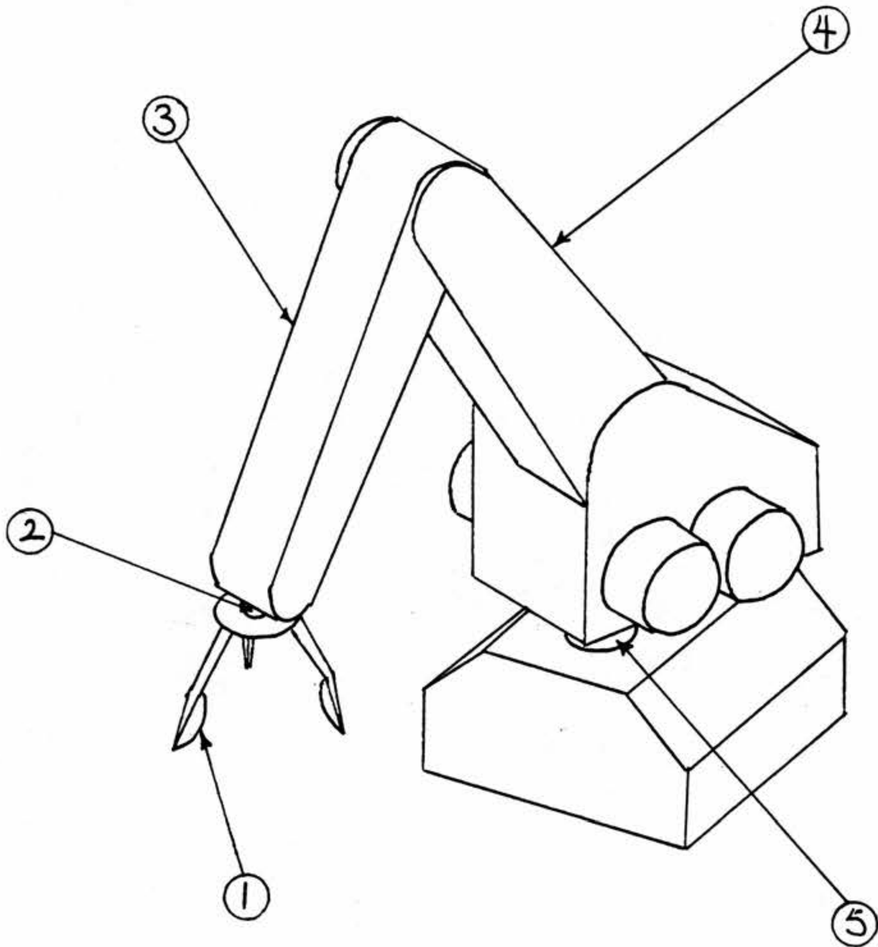


Figure 5.3 Armdroid

Component	Part No.
Finger	1
Wrist	2
Forearm	3
Upper arm	4
Shoulder	5

Scale 1:5 (approximate)

5.4.3 Hardware connection module

Four types of hardware module: Buffer Box, DC motor module, Stepping motor module and Sensor module, were built to facilitate connection between the computers and the control devices. These modules were designed and built by the author with the help of the technical staff from the Department. The Buffer box, DC motor module and Sensor module are the same as the ones described in section 2.1.3. Figure 5.4 shows the Stepping motor module. There are six DIN sockets on the top of the module. The sockets are numbered 1 to 6. If a stepping motor is plugged into socket N then it will be referred to as MOTOR N in Concurrent-Logo. For example, the command

```
MOTOR 1! TURNC 200
```

would make the stepping motor connected to socket 1 turn clockwise 200 steps.

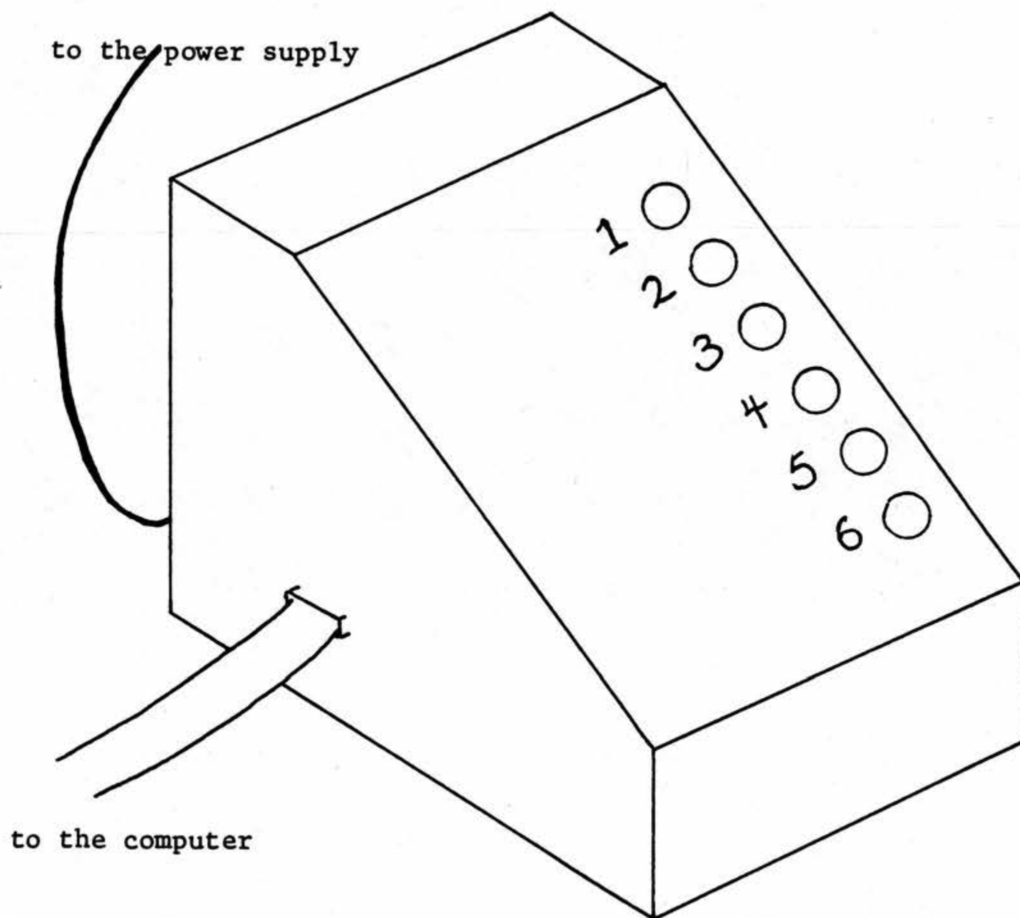


Figure 5.4 Stepping motor module

Scale 1:3 (approximate)

5.5 Course

Both year groups followed the same course. It consisted of a collection of six projects. Each project involved the students in writing programs for a particular control device decribed previously. As mentioned in chapter 1, the course was designed to

(1) give the students practical experience in using the basic control concepts: state, feedback and pulsing

(2) give the students practical experience in using programming constructs such as: procedures, conditionals and parallel processing.

(3) help the students understand how the devices work

(4) familarise the students with components which are commonly used in control devices: DC-motor, stepping motor, button switch, reed switch, reflective-opto switch and micro switch

5.5.1 Design principles

The principles for designing the collection of control devices were:

(1) the devices should interest the students.

(2) the purposes and functions of the devices should be apparent to the students.

(3) the students should be able to do interesting applications with the devices without having to get over many initial hurdles

(4) the devices should provide scope for project ideas. Individual students should be able to try out ideas up to their own level of competence.

(5) the devices should form a coherent set, starting from simple

to more elaborate ones, and they should cover a wide range of applications and control concepts.

Principles (1) to (3) were concerned with motivation. Principles (4) and (5) were concerned with the content of the course. Something which is enjoyable and interesting does not necessarily have educational value, so attention was given to the range of devices and the programming tasks covered by the course.

5.5.2 Course content

The content of the course is based on six projects. The description of each project includes

(1) the objectives of the project.

(2) a series of programming activities

(3) the control and related computing concepts in each of the activities.

5.5.2.1 Windmill

Objective:

To introduce the students to the ideas of sending signals, detecting signals, controlling motors and responding to signals.

Activity	Control concept	CLogo command
Use direct commands to make the windmill turn in different directions.	controlling a motor; sending signals	SWITCH command
Use direct commands to make the computer detect signals from a button-switch.	receiving signals	RECEIVER command
Use direct commands to make the computer detect signals continuously.	continuous monitoring	RECEIVER command; FOREVER command
Make the computer detect signals from three button-switches continuously so that the windmill will turn clockwise, turn anti-clockwise or stop depending on which of the button-switch was pressed most recently.	relating input and output	SWITCH command; RECEIVER command; IF command; FOREVER command

Figure 5.5 Windmill project

5.5.2.2 Turtle

Objective

To introduce the students to the ideas of coordinating the movement of two motors and controlling the speed of motors.

Activity	Control Concept	CLogo command & computing concepts
Define procedures FORWARD, BACKWARD, LEFT and RIGHT	Co-ordinating two motors	SWITCH command
Define procedure SWITCH-CONTROL so that the turtle may be controlled to move in different directions simply by pressing button-switches.	application of signal detection: control box	IF command; FOREVER command
Extend the SWITCH-CONTROL so that the speed of the turtle may be altered.	pulsing	REPEAT command; variable

Figure 5.6 Turtle I project

5.5.2.3 Lift

Objective

To introduce the ideas of current state of a device, list processing parallel processing and inter-process communication.

Activity	Control concepts	CLogo command & computing concepts
Use direct commands to make the lift move to different floors.		SWITCH n ONUNTIL
Define procedure GOTO so that the command GOTO :N will make the the lift move to the Nth floor.	current position; controlling a motor	IF command; variable; procedure with input
Define procedure SWITCH-CONTROL so that the lift may be controlled to move to different floors simply by pressing button-switches.	application of signal detection: control box	IF command; FOREVER command
Improve SWITCH-CONTROL so that while the lift is moving, signals should still be detected, recorded and processed in due course	scheduling; synchronization	list processing; parallel processing

Figure 5.7 Lift project

5.5.2.4 **Doll's house**

Objective

To show how a computer controlled security system works. To reinforce many of the concepts that the students would have learned from previous work.

Activity	Control concepts	CLogo command & computing concepts
Define procedure WINDOW so that when a window is being opened warning signals are given.	application of signal detection: burglar alarm system	IF command; FOREVER command
Define procedure DOOR so that the computer acts like a door keeper. Whenever the door bell is pressed the computer asks for a secret word or sentence. If the answer is correct the door slides open, otherwise the door remains closed.	application of signal detection: position sensing; door keeper; password	ASK command; list processing
Define procedure HOUSE so that both of the above ideas are combined in one program.		parallel processing

Figure 5.8 Doll's house project

5.5.2.5 **Turtle with optical sensors (Turtle II)**

Objective

To introduce the use of stepping motors and the idea of feedback.

Activity	Control concepts	CLogo command & computing concepts
Make procedures FORWARD, BACKWARD, LEFT and RIGHT.	co-ordinating two motors	MOTOR commands; parallel processing
Define procedure WALK so that the turtle will follow a track.	following a path in a set fashion	
Define procedure TRACK so that the turtle will follow a track making use of feedback information.	application of signal detection: feedback	FOREVER command; IF command
Define procedures that would make the turtle recognise binary coded patterns.	application of signal detection: pattern recognition	binary codes; IF command

Figure 5.9 Turtle II project

5.5.2.6 Robot arm

Objective

To introduce different methods of programming a robot, and the ideas of object collision, absolute position, and relative position.

Activity	Control concepts	CLogo command & computing concepts
Operate the arm using single key-presses.	robot movements	
Teach the arm a sequence of actions and then replay it.	programming robot through teaching; object collision	
Program the robot arm using MOTOR commands directly.	absolute position; relative position;	MOTOR commands; representing an arm position as a list of numbers

Figure 5.10 Robot arm project

5.5.3 Teaching method

As explained in Chapter 1 and 3, a structured framework was adopted for teaching control applications since it was felt that this would help the students to acquire a common vocabulary and set of concepts very quickly and would initiate them into thinking about different kinds of strategy for controlling a device.

For every project the author gave an introduction and some demonstrations to the students. The central classroom activity was the students programming the computers. Worksheets with a series of suggested programming tasks were provided for the students and they received help from the author whenever they asked. The author also

took the initiative and had discussions with the students individually and helped them correct programming errors.

5.5.4 Timetable

The fourth year group had seventeen sessions; the third year group had twenty four sessions (see figures 5.11 and 5.12).

Session 1 - 2	Session 3 - 4
Device: Windmill Worksheet: 1 - 4 Topic: SWITCH command; RECEIVER command; DC motor; IF command.	Device: Turtle I Worksheet: 5 - 6 Topic: Procedure; Turtle.

Session 5

Device:
Worksheet: 7 - 8
Topic: Arithmetic; variable; list.

Session 6 - 9 (Two devices in parallel)

Device: Lift Worksheet: 9 Topic: Lift	Device:Doll's house Worksheet: 10 Topic: Computer controlled security system.

Session 10

Intermediate progress survey

Session 11 - 16 (Two devices in parallel)

Device: Turtle II Worksheet: 11 - 13 Topic: Stepping motor; Reflective opto- switch; Binary code.	Device: Robot arm Worksheet: 14 - 16 Topic: A teaching program for the robot arm; How the arm works.

Session 17

Final survey and test.

Figure 5.11 Fourth year group time table

Session 1 - 2	Session 3 - 4
Device: Windmill Worksheet: 1 - 4 Topic: SWITCH command; RECEIVER command; DC motor; IF command.	Device: Turtle I Worksheet: 5 - 6 Topic: Procedure; Turtle.

Session 5 - 6

Device:
Worksheet: 7 - 8
Topic: Arithmetic; variable; list.

Session 7 - 11 (Two devices in parallel)

Device: Lift Worksheet: 9 Topic: Lift	Device:Doll's house Worksheet: 10 Topic: Computer controlled security system.

Session 12

Inermediate progress survey

Session 13 - 22 (Two devices in parallel)

Device: Turtle II Worksheet: 11 - 13 Topic: Stepping motor; Reflective opto-switch; Binary code.	Device: Robot arm Worksheet: 14 - 16 Topic: A teaching program for the robot arm; How the arm works.

Session 23 - 24

Final survey and test.

Figure 5.12 Third year group time table

Note: after a few sessions, the arrangement of sharing one device between two computers was found to be inconvenient. Therefore from session 6 (fourth year group) and session 7 (third year group) two different devices were introduced at once so that each computer had a device dedicated to it.

5.6 Evaluation

In this study, the illuminative evaluation approach (Parlett and Hamilton, 1977) was used. The approach is most suitable for studying innovatory programs. Its objective 'is to provide a comprehensive understanding of the complex reality (or realities) surrounding the program: in short, to "illuminate". In his report, therefore, the evaluator aims to sharpen discussion, disentangle complexities, isolate the significant from the trivial, and to raise the level of sophistication of debate.' The advantage of this approach over the pre- post-tests method is that it does not require a large sample size for the result to be valid. Furthermore, numerical results do not provide all the kinds of information that are relevant to this study, for example, information about why students make certain mistakes and how they solve problems.

An illuminative evaluation is characterised by three overlapping phases: observation, inquiry and explanation. 'The observation phase occupies a central place in illuminative evaluation. The investigator builds up a continuous record of ongoing events, transactions and informal remarks. At the same time he seeks to organize this data at source, adding interpretative comments on both manifest and latent features of the situation.' (Parlett and Hamilton, 1977). As the investigator becomes enlightened, he then directs his inquiry more systematically and forms more focussed questions. The final phase consists in finding general principles underlying the program being studied and finding explanation for certain observed trends.

This three-phase methodology was followed in the study. A record of all the sessions was kept. The data were analysed to help form focussed questions. Questionnaires and test were produced and used to gather further information. Finally, general conclusions were drawn.

During each session the following methods were used to collect data:

(1) observation. The author closely observed what was going on in the classroom, paying particular attention to the children's approaches to problem solving, their reaction to the computer's responses and their gradual familiarization with a piece of control equipment.

(2) dribble file. The Concurrent-Logo system automatically recorded all the commands issued to it on a dribble file. A copy of the procedures defined by the children was also kept on the disk.

(3) reflective questions. Questions like 'Why did you do it?' or 'Could you have done it in another way?' were asked to find out the intentions of a student.

Throughout the study, four questionnaires and a test (Appendix II) were designed and used. The first two questionnaires were for finding some background information about the students. The third questionnaire was an intermediate progress survey. It was used at the beginning of the second term. The fourth questionnaire was the final progress survey. The last questionnaire and the test were filled in by the students at the end of the course.

Questionnaires and test, instead of semi-structured interviews, were used because they were easier to conduct and to a certain extent guarded against the prejudice of the evaluator. Questionnaires and test provide a less sensitive measure of result but it was supplemented by detailed analysis of the students' work.

The outcome of the evaluation is in two parts. The first is a formative assessment of the work done by the students. It identifies what the students were capable of doing, the difficulties they faced and the differences within their work. The second is a general assessment of the study. It assesses the students' attitude towards the course, the students' achievement from the course and Concurrent-Logo.

CHAPTER 6

LEARNING CONTROL APPLICATIONS THROUGH PROGRAMMING

This chapter describes the work done by four of the students during the pilot study. The four students, two from the third year and two from the fourth year, are of varying abilities. Their work is fairly representative of the work done by all the students.

Section one provides some information on the four students. Section two summarises the work done by them and describes the difficulties that they faced. Sections three to eight give details of their work. The description is divided into projects and is in the order in which they were presented to the students. For each project, the description concentrates on:

(1) the variation in the students' work

(2) the difficulties that they faced

(3) how they solved the problems.

This chapter ends with a summary of the benefits and limitations of learning control applications through programming and identifies the practical problems of managing a course involving lots of equipment.

6.1 The students

None of the students had any previous experience of programming control devices.

Nigel is a fourth year student. He is very intelligent. His teacher rated him as well above average. He had a lot of programming experience in BASIC. He was very enthusiastic throughout the course.

In all the projects he was able to do more than any other student.

Heath is a fourth year student. His teacher rated him as a below average student. He had no computing experience prior to the course but he has a strong interest in computing. He is dominant in character. During the course he made most of the decisions for himself and for his partner, though his partner was a more able student. He had difficulties in getting started on a problem. He had to be told very carefully what was required of him.

Michael is a third year student. He had no computing experience. However, he is very intelligent and very keen to learn. He learned how to solve difficult problems very quickly.

Gary is a third year student. He is of average ability. He is the problem student of the four. He had an inflated assessment of his ability. Although he showed a poor understanding of his work, he always thought that the projects were easy. Hence, very often he needed to be persuaded to try them.

6.2 An overview

6.2.1 Understanding and appreciating problems

Nigel and Michael, the able students, had no difficulties in understanding and appreciating any of the problems posed to them. They readily accepted the challenge to solve the problems. Nigel deliberately skipped several parts that he genuinely found too easy, however.

Heath and Gary, the less able and average students, were easily put off by difficult problems. There are several reasons why they did

not want to try anything difficult:

(1) they were content if they could just make a device move.

(2) they did not care how clumsy a method was, as long as it met the end.

(3) they did not appreciate the difference between apparently similar problems and hence did not see why different solutions needed to be found to meet similar ends.

The first two reasons are related to attitude. Gary and Heath cared very little about quality or style; their main concern was to produce a desired effect. The third reason, however, is due to a limited understanding of automation. Gary's excuse points particularly to the third reason. The recurring excuse that he gave for not attempting a problem was: 'I can do that already, why find another way of doing it!' For example, after he had used direct commands to manipulate the windmill he did not appreciate why he should write a procedure for it. Other examples are: he did not want to write a GOTO procedure for the lift because he could use direct commands to make the lift go to different floors; he did not want to write a responsive procedure for turtle II because he had written a set procedure to walk the track; he did not want to write procedures to drive the robot arm because he could use the provided procedures to manipulate the arm.

When Heath found a problem difficult, instead of persisting, he would avoid it. He left out parts of the lift project, the Turtle II project and the robot project.

The author used different strategies with Heath and Gary. When Heath avoided a problem the author just let him carry on doing

something which he felt comfortable with. Because he was quite a motivated person this strategy worked very well in practice. It helped him to consolidate what he had already learned and helped him to build up his confidence.

Gary was less motivated and easily distracted so when he made an excuse for not attempting a problem the author would persuade him to try it. Of all the attempts to help Gary, the most successful tactic was to help him to appreciate the difference between a set procedure and a responsive procedure for Turtle II.

The programming task was to write procedures to make the Turtle follow a track. Gary wrote a procedure that would make the turtle walk the track in a set fashion. The turtle had to start at a fixed point, follow the same path and finish at another fixed point. When he was asked to write a procedure that made use of feedback information he refused because he did not see the point in writing another procedure to make the turtle walk the same track. He had to be persuaded that the responsive procedure is significantly different from the one he had already defined. The author used the analogy of training a blind man to follow a street. One way is to show the blind man exactly how many steps forward, then how much to turn and then how many steps forward, and so on. With this method, the blind man has to memorize every detail and then trace it out. If he has to walk another street he has to go through the same process of learning and remembering every step that he should take. Another way of training the blind man is to teach him how to use a walking stick, which is a much more flexible solution. After every few steps that the blind man has taken he can use the walking stick to find out

whether he should carry on going forward or turn, and in which direction to turn. After this explanation Gary could see the difference and was willing to try.

6.2.2 Difficulties and errors

Some misconceptions that the students held and mistakes that they made can be clearly identified through their programming activities. This section gives an overview of these problems. Details are given in later sections where the students' work is described.

6.2.2.1 Controlling motors

To recap, a DC motor is controlled by two switches, one for switching the motor on and off, the other for changing the motor's direction of rotation. There is a total of four possible switch states (see figure 4.1 in chapter 4).

Although it is simple, Gary had a lot of difficulties when writing programs involved in controlling DC motors: making the windmill turn in different directions; making the lift move up and down; opening and closing the sliding door in the doll's house. He knew that a motor is controlled by two switches but he showed no awareness of their different functional purposes. The problem was at its worst when programming Turtle I, which has two DC motors mounted back to back. To make the turtle move in different directions he had to coordinate the movement of both of the motors. With two motors there are four switches and sixteen possible switch states. Instead of studying how the SWITCH commands relate to the motors' movements and how the motors' movement relate to the turtle, he just defined

the procedures FORWARD, BACKWARD, LEFT and RIGHT independently from one another. The end result was that

```
FORWARD  made the turtle turn left about the left wheel
BACKWARD made the turtle move forward
LEFT     made the turtle move forward
RIGHT    made the turtle move backward.
```

Another example is equally revealing. He was given the program

```
GOUP ´FLOOR;
@set direction switch so that the lift would move up
SWITCH 2! ON;
@switch the motor on until recached the specified floor
SWITCH 1! ONUNTIL EQU? ´ON RECEIVER :FLOOR! STATE
```

for controlling the lift, which takes an integer as input and moves the lift upward until it reaches the specified floor. When asked how he would modify the program to make the lift move down, instead of just setting SWITCH 2 to OFF, he gave the definition

```
GODOWN ´FLOOR;
SWITCH 1! ON;
SWITCH 2! ONUNTIL EQU? ´ON RECEIVER :FLOOR! STATE
```

It is clear that he knew he had to reverse ´something´ to make the lift move in the opposite direction but he could not identify it as the state of the direction SWITCH that has to be changed. Instead, he changed the number 1 to 2 and the number 2 to 1.

6.2.2.2 Pulsing

The problem is to program Turtle I to move at different speeds. Only Nigel had enough time to try it. This problem showed up a misconception that he held.

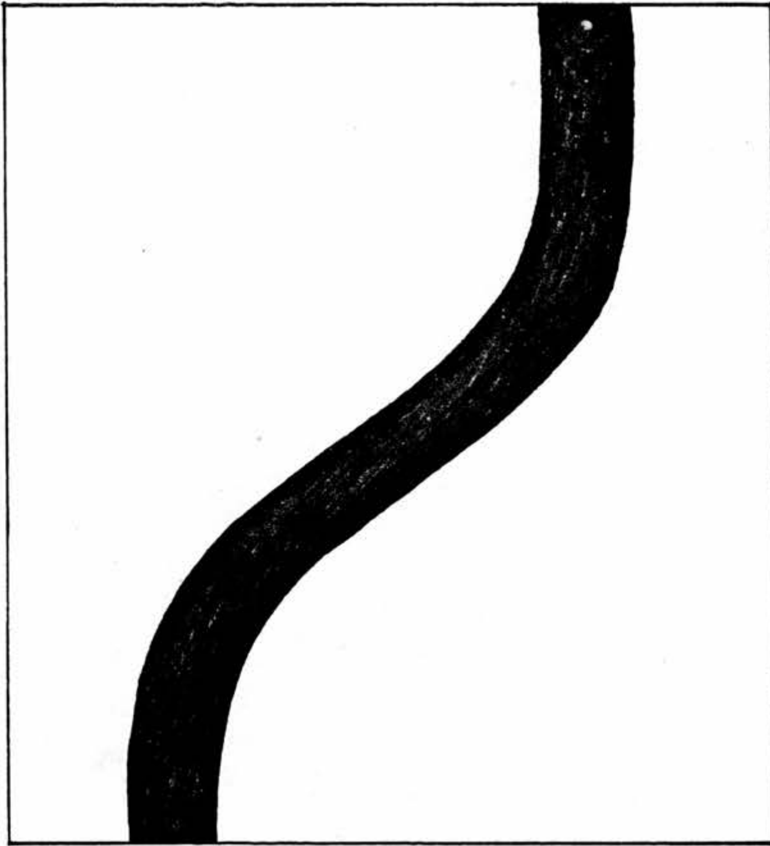
He thought that the speed of the turtle must be related to how 'busy' the computer was: the more work that the computer had to do the slower the turtle would move and vice versa. To have some control over the activities of the computer he used a repeat-loop. His idea was that by varying the repeat factor, which would vary the amount of time the computer spent executing the loop, the Turtle would move at different speeds. However, it was clear that the speed of the Turtle stayed constant. After some discussion with the author, Nigel realised that once a DC motor is switched on it stays on and rotates at a constant speed. The way to change its speed is by switching it on and off and varying the delay time.

6.2.2.3 GOTO procedure

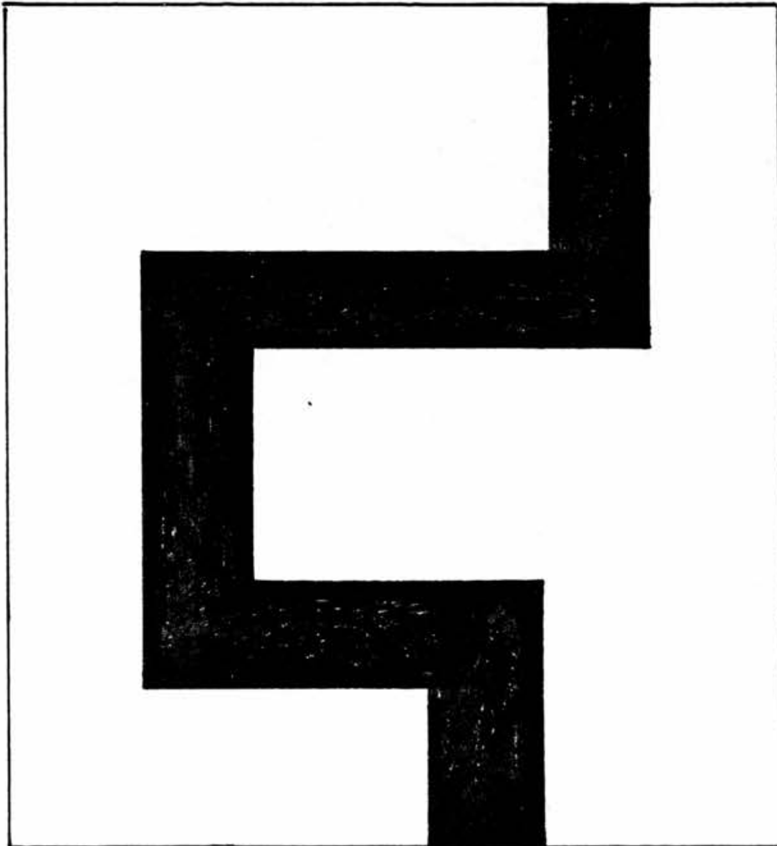
For all of the students, writing a procedure that takes a number as input and moves the lift to the corresponding floor was conceptually difficult. The problem lies in recognising and using the concept of current-state. Consider the case of moving a lift to the second floor. If the lift is currently at the first floor then it should move up; if it is currently at the third floor then it should move down. In other words, to move the lift to a particular floor a decision has to made with regard to the direction in which the lift should move. To make this decision the current position of the lift has to be taken into account. When controlling the lift in the direct mode the students made such decisions unconsciously. However, to automate the process they had the difficulty of identifying the steps which they themselves could execute effortlessly.

6.2.2.4 Following a track

The students were asked to write procedures for Turtle II to follow two tracks. The shapes of the tracks are shown in figure 6.1. A track is a black line, as wide as a turtle, painted on a white card board. When a turtle is directly on top of a track both reflective-opto switches would send an ON signal to the computer.



Track I



Track II

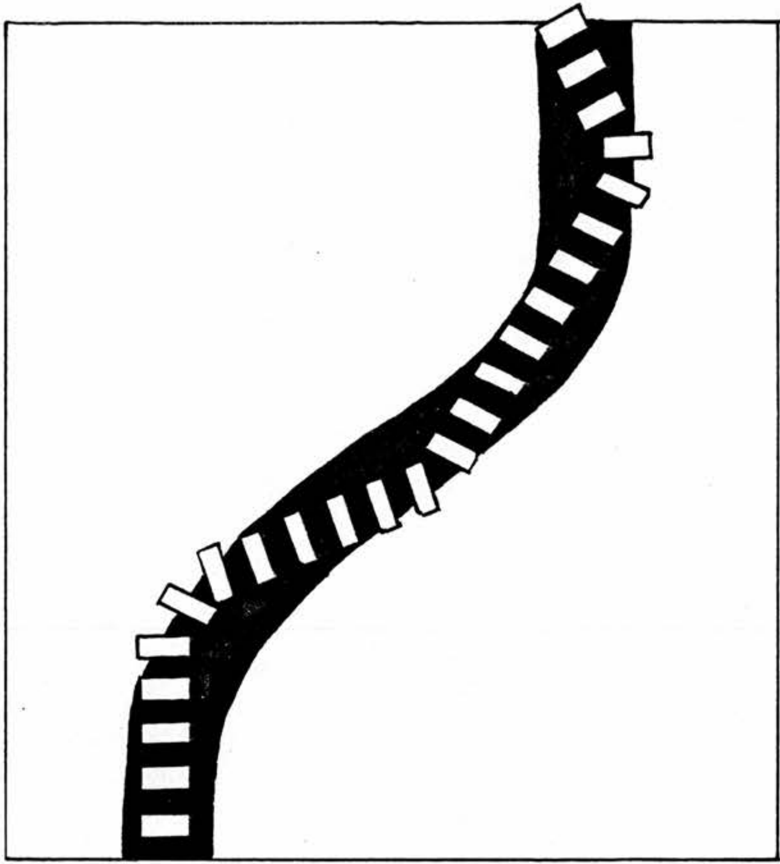
Figure 6.1 Turtle tracks

The students started by writing procedures that directed the turtle to follow a track in a set fashion. The turtle had to start at a fixed point, follow the same path and finish at another fixed point. Then they progressed to write responsive procedures, i.e. procedures that made use of the feedback information from the reflective-opto switches. They did not have much difficulty in formulating the control algorithm for following Track I. It is simply

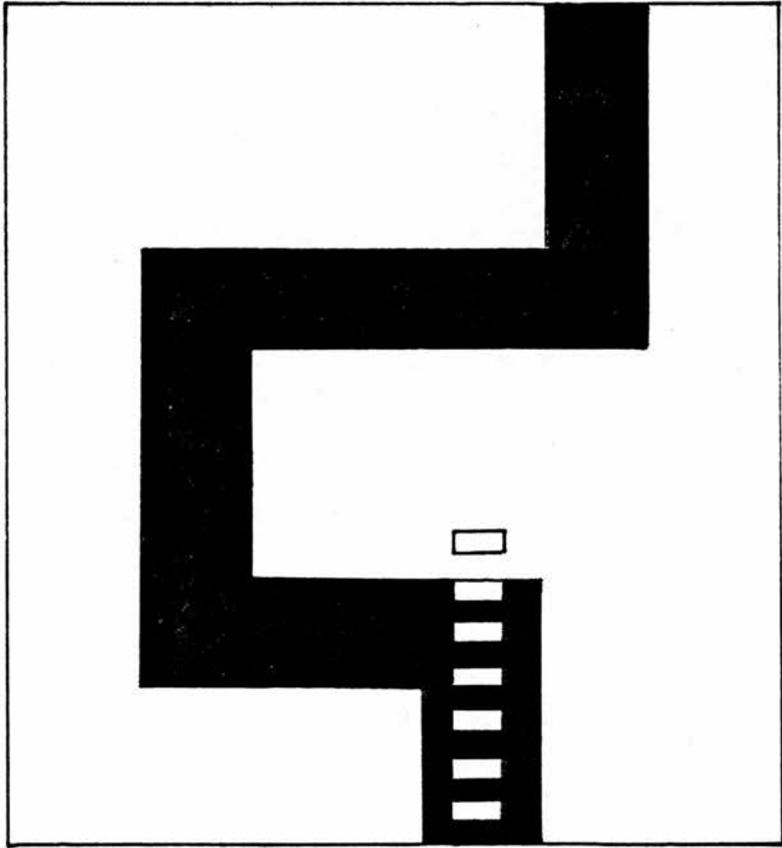
```
Forever (
1) forward a little
2) if left hand side is off the track turn right a little
3) if right hand side is off the track turn left a little
)
```

A typical turtle path is shown in figure 6.2 (Path I).

The students thought that the algorithm that they had derived for Track I would be sufficiently general for all tracks, including Track II. However, they were wrong. Using the same algorithm on track II the turtle went off the track and was unable to find its way back, illustrated in figure 6.2 (Path II).



Path I



Path II

Figure 6.2 Turtle paths

There are three aspects of the algorithm that had to be changed or extended:

(1) it should test whether the turtle is completely off the track

(2) if the turtle is completely off the track it should move backward onto the track before making any turn.

(3) the angle of rotation has to be increased to 90 degrees.

The students quickly realized that the angle of rotation had to be increased. However, the other two changes required were much more difficult to identify.

6.2.2.5 Pattern recognition

Another programming task for Turtle II was to define procedures for recognising binary-coded patterns. A pattern is represented by four bars. Each bar is either black or white. For example, the patterns shown in figure 6.3 were used to represent the letters ´A´ and ´B´. Fifteen patterns were made up to represent the letters ´A´ to ´O´. The author provided the students with a sample procedure, CODE.A, which confirms whether a pattern represents the letter ´A´.

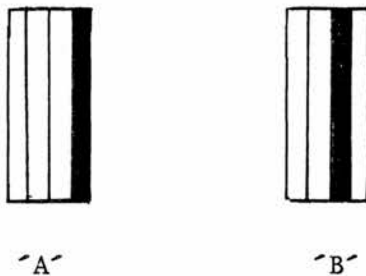


Figure 6.3 Binary patterns

The students were asked to modify the procedure to recognise other letters. Heath did not find the solution obvious at all. Although he could not understand the algorithm used in the given program, he started by changing the parts which were obvious. To modify CODE.A to recognise the letter ´B´, he renamed CODE.A as CODE.B and replaced the output messages [THE CODE IS ´A´] and [THE CODE IS NOT ´A´] by [THE CODE IS ´B´] and [THE CODE IS NOT ´B´] respectively. Though his initial modification did not work properly it gave him the interest and confidence to find out how the algorithm works. After several modifications he was able to change the given procedure to recognise the letters ´B´ and ´C´.

Michael did not find the above programming task difficult but had lots of difficulties in defining a procedure that could identify any of the fifteen patterns. His idea was to build fifteen procedures: CODE.A, CODE.B ... etc, to recognise all of the patterns and then think of a way of combining the procedures. It seemed hard work and he could not see how the procedures might be combined.

The author reminded him that the turtle need walk over the pattern only once, therefore, the computer should remember the pattern and then compare and decide which letter it represents. When the recognition algorithm was pointed out, translating it into program form was easy.

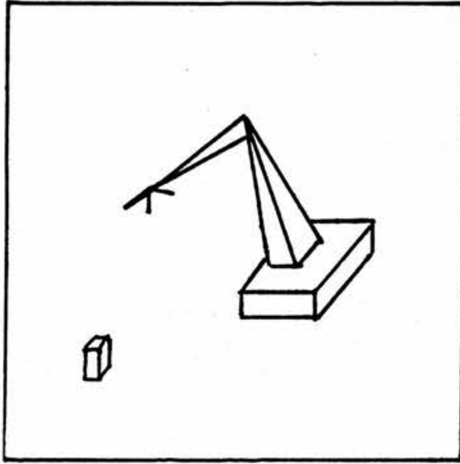
Recognising a simple pattern is an example of a task which a student is able to do very naturally without being aware of how he himself does it.

6.2.2.6 Teaching the robot arm

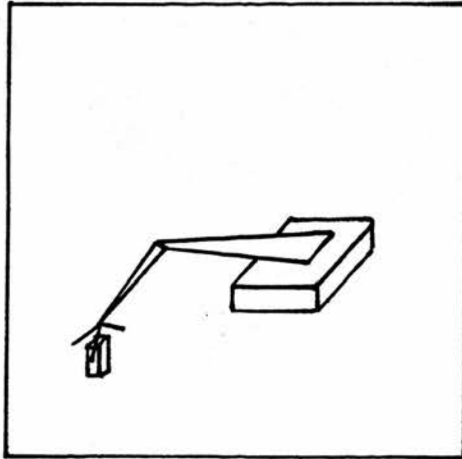
The students were given a set of procedures that allowed them to manipulate the robot arm using single key-presses, to request the computer to remember a sequence of arm positions, and to replay a movement.

The students did not have many problems in operating the arm. However, they all made a common mistake when teaching the arm to remember a sequence of movements. They did not identify all of the important arm positions that need to be remembered.

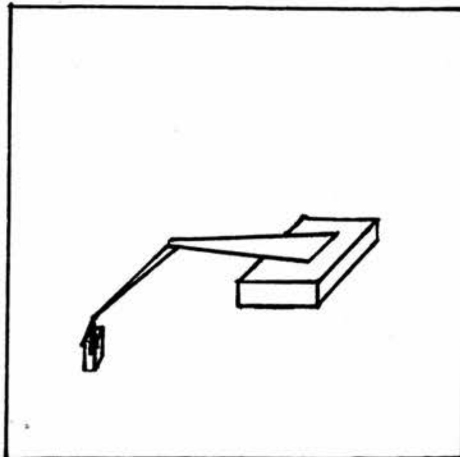
As an example, consider teaching the arm to pick up a block. A student would manipulate the arm, using the provided primitives. In the process the computer needs to be told to remember certain arm positions in order that the action can be replayed successfully. Figure 6.4 shows the initial position of the arm and the arm positions that must be remembered in the correct order.



Position 1



Position 2



Position 3

Figure 6.4 Sequence of robot arm positions (I)

When the students first programmed the robot arm they all missed out position 2. As a result, when the computer was asked to replay the action the arm moved directly from position 1 to 3. When the arm reached the block its fingers were already closed, so the arm knocked the block over instead of picked it up.

6.2.3 Summary of students' activities

The following tables summarise the work done by each of the four students during the course.

<u>Keys to the tables</u>

'/' means completed
'X' means tried but not completed
'-' means not tried

The number beneath each student's name is the number of minutes that he spent on the device.

	Nigel (35)	Heath (35)	Michael (35)	Gary (35)
Detecting signals	/	/	/	/
Sending signals	/	/	/	/
Relating input & output	/	/	/	/

The windmill

	Nigel (50)	Heath (50)	Michael (50)	Gary (50)
Basic procesures	/	/	/	X
Procedure SWITCH-CONTROL	/	/	/	/
Puulsing	/	-	-	-

The turtle

Note: the basic procedures are FORWARD, BACKWARD, LEFT and RIGHT.

	Nigel (40)	Heath (50)	Michael (60)	Gary (60)
Predefined procedures & direct commands	/	/	/	/
Procedure GOTO	/	-	/	X
Procedure SWITCH-CONTROL	/	/	/	/
Scheduling	/	-	-	-

The lift

	Nigel (40)	Heath (50)	Michael (60)	Gary (60)
Procedure WINDOW	/	/	/	/
Procedure PASSWORD	/	/	/	/
Procedure DOOR	/	/	/	-
Procedure HOUSE	/	/	/	/

The security system

	Nigel (75)	Heath (85)	Michael (100)	Gary (125)
Basic procedures	/	/	/	/
Procedure WALK	-	/	/	/
Procedure TRACK (I)	/	-	/	/
Procedure TRACK (II)	/	-	/	-
Procedure PATTERN (I)	-	/	/	-
Procedure PATTERN (II)	/	—	/	-

Turtle II

	Nigel (75)	Heath (85)	Michael (125)	Gary (125)
Operating	/	/	/	/
Teaching	/	/	/	/
Programming	/	-	-	-

Robot arm

6.3 Windmill

The windmill (see figure 2.2 in chapter 2) has one DC motor for spinning the sails. The motor was connected to SWITCHes 1 and 2. SWITCH 1 was the direction switch and SWITCH 2 was the power switch.

The objective of the project is to introduce the ideas of sending signals, detecting signals, controlling motors and responding to signals. Because the device is very simple there was not much variation among the students' work.

6.3.1 Sending signals

The students were introduced to the SWITCH commands. They were asked to experiment with changing the states of SWITCHes 1 and 2. They typed in commands to turn the SWITCHes on and off. There were expressions of satisfaction when the windmill turned. Later, they were also encouraged to experiment with SWITCHes other than 1 and 2. After a few commands, it was clear to them that turning on or off a SWITCH which had nothing connected to it had no effects.

6.3.2 Detecting signals

They were then introduced to the RECEIVER commands. A push-button switch was connected to RECEIVER 1. They were asked to detect whether the switch was pressed. They did this by using the command

```
PRINT RECEIVER 1! STATE.
```

The FOREVER command was also taught so that the state of a RECEIVER might be continuously monitored:

```
FOREVER (PRINT RECEIVER 1! STATE).
```

The students had great fun in pressing and releasing the switch and seeing the words ON and OFF printed on the screen accordingly. They had no problems in understanding the FOREVER and RECEIVER commands.

They were then encouraged to detect the states of RECEIVERs which had nothing connected to them. Again, it was clear to them that a RECEIVER would change state only if an input device was connected to it.

6.3.3 Relating input and output

Two additional push-button switches were connected to RECEIVERs 2 and 3. The idea was to make the computer detect signals from the RECEIVERs so that whenever:

(1) push-button switch 1 is pressed the windmill turns clockwise

(2) push-button switch 2 is pressed the windmill turns anticlockwise

(3) push-button switch 3 is pressed the windmill stops.

The IF command was an essential part of this project. Because the IF command is quite complicated, both syntactically and conceptually, a worksheet was produced. There was no resistance to the idea of following a worksheet.

By the end of the second session they all had come up with the correct sequence of commands:

```
FOREVER(
  IF EQU? ´ON RECEIVER 1! STATE (SWITCH 1! ON; SWITCH 2! ON);
  IF EQU? ´ON RECEIVER 2! STATE (SWITCH 1! OFF; SWITCH 2! ON);
  IF EQU? ´ON RECEIVER 3! STATE (SWITCH 2! OFF)
)
```

The windmill project was a simple and effective way of helping the students to

(1) understand the SWITCH, RECEIVER, FOREVER and IF commands

(2) appreciate the concept of sending and receiving signals

(3) appreciate that the computer can be used to control an external device

At the end of this project Nigel asked to be taught the editing facility in Concurrent-Logo, because it was very tedious and error-

prone to type in a long sequence of commands again and again. A worksheet was then prepared for introducing the students to the concept of procedures and the window editor.

6.4 Turtle I

The turtle (see figure 4.2 in chapter 4) has two DC motors mounted back to back on it. The right motor drives the right wheel and the left motor drives the left wheel. Figure 6.5 shows the motors´ connections.

Component	Concurrent-Logo object
DC motor (part 1)	SWITCH 1 (direction) SWITCH 2 (on/off)
DC motor (part 2)	SWITCH 3 (direction) SWITCH 4 (on/off)

Figure 6.5 Turtle I connection

The objective of the project was to introduce the ideas of coordinating the movement of two motors and controlling their speed.

6.4.1 FORWARD, BACKWARD, LEFT and RIGHT

Elimination: effective use of trial and error

Nigel showed considerable ability in identifying the relevant information to solve this task. He was the only student who spontaneously experimented with the turtle using direct commands before attempting to define the procedures. He systematically changed the states of the SWITCHes and tabulated the resulting

movement of the turtle. With that information he defined the procedures correctly at the first attempt.

The commands in his procedures were organised in an interesting way. In his first procedure, FORWARD, the sequence of SWITCH commands was in the ascending order of the SWITCH index numbers:

```
FORWARD;
SWITCH 1! ON; SWITCH 2! ON; SWITCH 3! OFF; SWITCH 4! ON
```

In his subsequent procedures the order of the SWITCH commands was changed

```
BACKWARD;
SWITCH 1! OFF; SWITCH 3! ON; SWITCH 2! ON; SWITCH 4! ON

LEFT;
SWITCH 1! ON; SWITCH 3! ON; SWITCH 2! ON; SWITCH 4! ON

RIGHT;
SWITCH 1! OFF; SWITCH 3! OFF; SWITCH 2! ON; SWITCH 4! ON
```

The author asked him why he had done that. He explained that this would set up correctly the motors' directions of rotation prior to making them move. He was clearly aware of the different functional purposes of the SWITCHes, i.e. SWITCHes 1 and 3 were used as direction switches, and SWITCHes 2 and 4 were used as on/off switches.

Michael also grouped the SWITCH commands according to their functional purposes. The difference between his procedures and Nigel's was that the motors were switched on before setting their direction of rotation. For example the definition of BACKWARD was

```
BACKWARD;
SWITCH 2! ON; SWITCH 4! ON; SWITCH 3! OFF; SWITCH 1! OFF
```

This definition has the disadvantage of making the turtle move in an undefined direction for a moment, depending on the states of SWITCHes 3 and 4 when the procedure is called. However, in practice the effect was not noticeable. Since Michael was fully aware of the different functional purposes of the SWITCHes he had no problem in defining the procedures.

The students were also asked to write a SWITCH-CONTROL procedure. Five button-switches were connected to RECEIVER 1 to RECEIVER 5. The purpose is that whenever

(1) switch 1 is pressed the turtle moves forward

(2) switch 2 is pressed the turtle moves backward

(3) switch 3 is pressed the turtle moves right

(4) switch 4 is pressed the turtle moves left

(5) switch 5 is pressed the turtle stops moving.

Both Nigel and Michael found this problem very easy and came up with the correct program:

```
SWITCH-CONTROL;
FOREVER (
   IF EQU? RECEIVER 1! STATE ^ON (FORWARD);
   IF EQU? RECEIVER 2! STATE ^ON (BACK);
   IF EQU? RECEIVER 3! STATE ^ON (RIGHT);
   IF EQU? RECEIVER 4! STATE ^ON (LEFT);
   IF EQU? RECEIVER 5! STATE ^ON (STOP)
)
```

Gary had a lot of difficulties in defining the four basic procedures. He defined FORWARD as

```
FORWARD;
SWITCH 1! ON; SWITCH 2! ON; SWITCH 3! ON; SWITCH 4! OFF
```

which had the left motor turned off. Instead of correcting FORWARD,

he went on to define BACKWARD, LEFT and RIGHT. He defined each of the procedures independently, without considering the relationship of all of the procedures. His procedure BACKWARD made the turtle move forward; LEFT made the turtle move forward; RIGHT made the turtle move backward.

Gary's SWITCH-CONTROL was defined correctly but its sub-procedures were not.

A subtle bug

The first procedure that Heath defined was

```
FORWARD;
SWITCH 1! ON; SWITCH 2! ON; SWITCH 4! ON
```

He deliberately left out the command SWITCH 3! OFF, which sets the direction of rotation for the left motor. The procedure worked because the initial state of SWITCH 3 was off. However, it was wrong to assume that the state of SWITCH 3 would be always OFF. The bug manifested itself when Heath was testing his SWITCH-CONTROL procedure. Although the definition of SWITCH-CONTROL was correct, the procedure did not work properly. Since FORWARD did not explicitly set SWITCH 3 to OFF, when button 1 was pressed the turtle sometimes moved forward and sometimes rotated left instead.

Heath redefined SWITCH-CONTROL twice and it still did not work. The author advised him to identify the particular case when SWITCH-CONTROL did not work. He then realized the bug might have been in FORWARD instead of SWITCH-CONTROL. After he had included the command SWITCH 3! OFF in FORWARD, everything worked properly.

Turning about one wheel vs Turning on the spot

Another interesting variation is the way that Heath initially defined the procedure LEFT:

```
LEFT;
@rotate right wheel in the forward direction
SWITCH 1! ON; SWITCH 2! ON;
@left wheel stationary
SWITCH 4! OFF
```

The author asked him whether he thought the procedure would work. He confidently said, ´Yes!´ When he ran the procedure, indeed the turtle did turn left. With the left motor turned off the turtle rotated left about its left wheel. Then the author asked if there were other ways of making the turtle turn left. Heath did not think so. The author showed him that the turtle could also turn left about its centre. Heath exclaimed, ´This is turning on the spot!´ The author explained that it was turning left on the spot and the turtle could also turn right on the spot. Heath re-defined LEFT and then defined RIGHT to make the turtle turn left and right on the spot respectively.

6.4.3 Pulsing

Only Nigel had time to modify the SWITCH-CONTROL program to make the turtle move at different speeds. In this extension he used two additional button-switches. One was connected to RECEIVER 6. It was for sending signals to decrease the speed of the turtle, referred to below as decrease-speed switch. The other was connected to RECEIVER 7. It was for sending signals to increase the speed of the turtle, referred to as increase-speed switch.

As described in section 6.2.2, Nigel thought that the speed of the turtle was related to how ´busy´ the computer was. He introduced a repeat-loop in his program. The number of times that the loop is executed is determined by the value of a variable ´X. Whenever the decrease-speed switch was pressed the value of ´X would be incremented by one, so that the computer would spend more time in processing the loop, and whenever the increase-speed switch was pressed the value of ´X would be decremented by one. The loop was a means of controlling the computer´s processing activities.

The definition of the procedure is

```
SPEED;
MAKE ´X 10;
FOREVER(
   IF EQU? RECEIVER 6! STATE ´ON (MAKE ´X ADD :X 1);
   IF EQU? RECEIVER 7! STATE ´ON (MAKE ´X SUB :X 1);
   SWITCH-CONTROL
)


SWITCH-CONTROL;
IF EQU? RECEIVER 1! STATE ´ON (FORWARD);
IF EQU? RECEIVER 2! STATE ´ON (BACK);
IF EQU? RECEIVER 3! STATE ´ON (RIGHT);
IF EQU? RECEIVER 4! STATE ´ON (LEFT);
IF EQU? RECEIVER 5! STATE ´ON (STOP);
REPEAT :X ();
```

When Nigel tried the program, the turtle moved at exactly the same speed as before. It did not respond when either the increase-speed switch or the decrease-speed switch was pressed. However, the response time between pressing a switch and the turtle changing its direction of movement had increased.

After some discussion with the author, Nigel concluded that

(1) the more activities that the computer has to attend to the

slower is the response time and vice versa,

(2) the computer´s response time does not affect the speed of a DC motor.

Nigel then focused his attention on slowing down the motors and formulated the idea of pulsing - switching the motors on and off.

He kept SPEED the same and modified SWITCH-CONTROL to

```
SWITCH-CONTROL;
IF EQU? RECEIVER 1! STATE ´ON (FORWARD);
IF EQU? RECEIVER 2! STATE ´ON (BACK);
IF EQU? RECEIVER 3! STATE ´ON (RIGHT);
IF EQU? RECEIVER 4! STATE ´ON (LEFT);
IF EQU? RECEIVER 5! STATE ´ON (STOP);
REPEAT :X (SWITCH 2! OFF; SWITCH 4! OFF);
SWITCH 2! ON; SWITCH 4! ON
```

The final program was a success in the sense that he could control the turtle´s speed. On the other hand, he had introduced a bug into the program: even if the turtle was stopped, by pressing button-switch 5, it would always start moving again voluntarily. The last two commands in SWITCH-CONTROL always switch the motors on.

6.5 Lift

The lift (see figure 4.4 in chapter 4) has one DC motor and three reed switches. The motor drives the lift cage up and down. The reed switches detect whether the lift cage is at a particular floor. Figure 6.6 shows these components´ connections.

Component	Concurrent-Logo object
Reed switch (part 1)	RECEIVER 1
(part 2)	RECEIVER 2
(part 3)	RECEIVER 3
DC motor (part 4)	SWITCH 1 (on/off) SWITCH 2 (direction)

Figure 6.6 Lift connection

The objective of the project was to introduce the ideas concerning the current state of a device. For advanced students the ideas of parallel processing and inter-process communication could also be taught.

6.5.1 Making the lift move

The author defined an object LIFT in Concurrent-Logo. It can respond to three commands: READY, UPANDDOWN and MOVETO. The idea of these commands was to help the students to become familiar with the lift and encourage them to find out how it works. READY moves the lift cage to the first floor and initialises a variable to the value 1. UPANDDOWN moves the lift to the top (third) floor and down to the first floor again. MOVETO takes a number as input and moves the lift to the specified floor. If the number is not between one and three nothing is done.

Michael was quite inventive. He used REPEAT loops to simulate the time that a lift spent waiting at a floor.

He defined a procedure LIFT as

```
LIFT;
LIFT! MOVETO 2; REPEAT 1000 (PRINT [HURRY UP]);
LIFT! MOVETO 1; REPEAT 100  (PRINT [QUICKLY]);
LIFT! MOVETO 3; REPEAT 100  (PRINT [RUN]);
LIFT! MOVETO 1
```

Michael could obviously relate the model lift to lifts that he was familiar with.

After the students had spent about ten minutes playing with the provided procedures they were asked to operate the lift using SWITCH commands directly. They were also taught the command

```
SWITCH N! ONUNTIL <condition>
```

An example of its use is

```
@set direction switch to move lift up
SWITCH 2! ON
@start lift moving until it reaches the third floor
SWITCH 1! ONUNTIL EQU? 'ON RECEIVER 3! STATE
```

After the students had operated the lift using direct commands they all made comments such as, 'Now I know how a lift works.'

6.5.2 GOTO and SWITCH-CONTROL

The next part of the project was to make the computer detect signals from three button switches so that whenever

(1) button switch 1 (connected to RECEIVER 4) was pressed the lift moved to the first floor.

(2) button switch 2 (connected to RECEIVER 5) was pressed the lift moved to the second floor.

(3) button switch 3 (connected to RECEIVER 6) was pressed the lift

moved to the third floor.

The students found this problem very difficult. Only Nigel was able to complete it without help.

Isolation: isolate difficulties

Instead of defining a GOTO procedure that takes a number as input, Nigel defined three separate procedures GOTO1, GOTO2 and GOTO3. He realised that to make the lift move to the bottom floor the lift should move downward, and to make the lift move to the top floor the lift should move upward. The difficulty is to make the lift move to the second floor. Nigel thought that splitting a general procedure into three more specific ones would help him to concentrate on solving the particular problem.

He defined GOTO1 and GOTO3 correctly at the first attempt:

```
GOTO1;
@set direction switch to move lift down
SWITCH 2! OFF;
@move the lift to the first floor
SWITCH 1! ONUNTIL EQU? RECEIVER 1! STATE ´ON;
@remember the lift is at first floor
MAKE ´LOCATION 1
```

```
GOTO3;
@set direction switch to move lift up
SWITCH 2! ON;
@move the lift to the third floor
SWITCH 1! ONUNTIL EQU? RECEIVER 3! STATE ´ON;
@remember the lift is at third floor
MAKE ´LOCATION 3
```

However, it took him several tries to work out exactly how to define GOTO2:

```
GOTO2;
@if the lift is at the top floor, set the lift to move down
IF EQU? :LOCATION 3 (SWITCH 2! OFF);
@if the lift is at the bottom floor set the lift to move up
IF EQU? :LOCATION 1 (SWITCH 2! ON);
@move the lift to the second floor
SWITCH 1! ONUNTIL EQU? RECEIVER 2! STATE ´ON;
@remember the lift is at the second floor
MAKE ´LOCATION 2
```

With the three GOTO procedures, Nigel easily re-defined SWITCH-CONTROL as

```
SWITCH-CONTROL
@Initialisation:
@move the lift to the first floor
SWITCH 2! OFF;
SWITCH 1! ONUNTIL EQU? RECEIVER 1! STATE ´ON;
@ remember the lift is at the first floor
MAKE ´LOCATION 1;
@main control loop
FOREVER(
   IF EQU? RECEIVER 4! STATE ´ON (IF NOT EQU? :LOCATION 1 (GOTO1));
   IF EQU? RECEIVER 5! STATE ´ON (IF NOT EQU? :LOCATION 2 (GOTO2));
   IF EQU? RECEIVER 6! STATE ´ON (IF NOT EQU? :LOCATION 3 (GOTO3))
)
```

Teacher´s guidance

Michael was not sure how to approach the GOTO procedure. The author helped him by asking him to define two simpler procedures that would help him to appreciate where the difficulties lie. The procedures are GOUP and GODOWN.

```
GOUP ´N;
SWITCH 2! ON;
SWITCH 1! ONUNTIL EQU? ´ON RECEIVER :N! STATE

GODOWN ´N
SWITCH 2! OFF;
SWITCH 1! ONUNTIL EQU? ´ON RECEIVER :N! STATE
```

GOUP takes a number as input and moves the lift upward until it reaches the specified floor; GODOWN takes a number as input and moves

the lift downward until it reaches the specified floor.

The author then helped him to explore these two procedures. Starting with the lift at the first floor, the author asked him to make the lift move to the second floor. He typed 'GOUP 2' and pressed RETURN. When the lift stopped at the second floor the author then asked him to make the lift move to the third floor, he typed 'GOUP 3' and pressed RETURN. When the lift stopped at the third floor, the author asked Michael if he would type GOUP 2 to make the lift move to the second floor. He said, 'No.' Michael typed GODOWN 2 and pressed RETURN. The author then asked Michael to make the lift move to the first floor. Michael typed GODOWN 1 and pressed RETURN. When the lift stopped at the first floor, the author asked Michael if he would type GODOWN 2 to make the lift move to the second floor. He said, 'No' and it suddenly dawned on him how to define the procedure GOTO. The procedure is

```
GOTO 'N;
IF LESS? :N :CURRENT (GODOWN :N);
IF GRE?  :N :CURRENT (GOUP   :N);
MAKE 'CURRENT :N
```

Once the GOTO procedure was defined he had no problem in defining SWITCH-CONTROL.

Gary also had trouble in defining GOTO. The author used the same strategy that he had used to help Michael. Because Gary has difficulties with DC motors, he could not even define GOUP. So, the author defined GOUP for him and asked him to modify it for GODOWN. As described in section 6.2.2, Gary knew that GODOWN is opposite to GOUP. But, instead of changing the motor's direction of rotation by switching SWITCH 2 off, he replaced the number 2 by 1 and 1 by 2:

```
GODOWN 'FLOOR;
SWITCH 1! ON;
SWITCH 2! ONUNTIL EQU? 'ON RECEIVER :FLOOR! STATE
```

He had to redefine GODOWN five times before he could get it right. He did not have enough time to carry on the project further.

Heath avoided the problem and took a familiar path. He defined SWITCH-CONTROL such that whenever

(1) switch 1 is pressed the lift moves upward

(2) switch 2 is pressed the lift moves downward

(3) switch 3 is pressed the lift stops moving.

Heath also defined a procedure that made the lift move to the third floor then down to the first and up to the second.

```
LIFT;
SWITCH 2! ON;
SWITCH 1! ONUNTIL EQU? RECEIVER 3! STATE 'ON;
SWITCH 2! OFF;
SWITCH 1! ONUNTIL EQU? RECEIVER 1! STATE 'ON;
SWITCH 2! ON;
SWITCH 1! ONUNTIL EQU? RECEIVER 2! STATE 'ON
```

Although he knew how to make the lift move up and down and how to detect whether the lift had reached a particular floor, he was not able to specify the GOTO algorithm.

6.5.3 Scheduling

The next extension was to improve on the SWITCH-CONTROL procedure so that it would not ignore signals from the button switches while the lift was moving from one floor to another. Only Nigel had time to do this part of the project.

The author asked him how the problem might be solved. Nigel recognised that the procedure needed to be changed so that it switched back and forth between detecting whether the lift had arrived at the specified floor and detecting signals from the button switches. He was unwilling to implement the change because the control flow of the procedure would be very complicated. The author suggested that the solution would be simpler if he had used parallel processing.

The author spent some time teaching Nigel about running procedures in parallel and explaining the multi-programming solution to the lift problem. However, the two problems related to parallel processing, mutual exclusion and synchronization, were not mentioned.

Nigel defined two new procedures

```
LIFT;
SWITCH // BUTTON

BUTTON;
MAKE ´LIST [];
FOREVER(
   @if a signal from a switch is detected,
   @put the request into a list
   IF EQU? RECEIVER 4! STATE ´ON (MAKE ´LIST PUTL 1 :LIST);
   IF EQU? RECEIVER 5! STATE ´ON (MAKE ´LIST PUTL 2 :LIST);
   IF EQU? RECEIVER 6! STATE ´ON (MAKE ´LIST PUTL 3 :LIST)
)
```

and modified SWITCH-CONTROL to

```
SWITCH-CONTROL;
@Initialisation
@move the lift to the first floor
SWITCH 2! OFF;
SWITCH 1! ONUNTIL EQU? RECEIVER 1! STATE ´ON;
@remember the lift is at the first floor
MAKE ´LOCATION 1;
@main control loop
FOREVER(
   @get the next request from a list
   MAKE ´X FIRST :LIST;
   @update the list
   MAKE ´LIST REST :LIST;
   @move the lift accordingly
   IF EQU? :X 1 (IF NOT EQU? :LOCATION 1 (GOTO1));
   IF EQU? :X 2 (IF NOT EQU? :LOCATION 2 (GOTO2));
   IF EQU? :X 3 (IF NOT EQU? :LOCATION 3 (GOTO3))
)
```

When Nigel had completed these procedures he was very pleased with the elegance of using parallel processing. Unfortunately his procedures did not work properly because there was a subtle bug in the first line of the FOREVER loop of SWITCH-CONTROL. It assumed that the list of requests was never empty. It would have been a good opportunity to discuss process synchronization with Nigel but there was no more time for this project. Up to this point, only Nigel had used parallel processing.

6.6 Doll´s house

The doll´s house (see figure 4.3 in chapter 4) has one DC motor for opening and closing the sliding door, two reed switches for detecting the closed and open positions of the door, four micro-switches, one behind each window, and one button switch, used as a door bell. Figure 6.7 shows these components´ connections.

Component		Concurrent-Logo object
Reed switches	(part 1)	RECEIVER 1
	(part 2)	RECEIVER 2
Micro switches	(part 3)	RECEIVER 3
	(part 4)	RECEIVER 4
	(part 5)	RECEIVER 5
	(part 6)	RECEIVER 6
Button switch	(part 7)	RECEIVER 7
DC motor	(part 8)	SWITCH 1 (on/off) SWITCH 2 (direction)

Figure 6.7 Doll's house connection

The objective of the project was to introduce how a computer could be used to protect a house against burglars, and to reinforce many concepts that the students would have learned from previous work.

6.6.1 WINDOW

The first task was to make the computer detect if burglars were trying to break in through any of the windows. If any of the windows was open, the computer should sound a continuous tone and print out a message telling exactly which window was opened. While the computer was sounding the alarm it should still be checking whether burglars were trying to get in through other windows.

All the students started the problem by using direct commands to confirm which RECEIVERs were used for detecting the states of the different windows.

The procedures defined by Nigel and Michael were very similar

```
WINDOW;
FOREVER(
   IF EQU? RECEIVER 3! STATE ´ON
      (PRINT [BREAK IN AT TOP RIGHT WINDOW];   SOUND);
   .....
   IF EQU? RECEIVER 6! STATE ´ON
      (PRINT [BREAK IN AT BOTTOM LEFT WINDOW]; SOUND)
)
```

The only difference is that in Nigel´s version four beeps, instead of one, are made when a window is being broken into. They both wanted to make the computer sound continuously and at the same time check whether any other window was being opened. Nigel was not prompted to use parallel processing. The solution that they both adopted was to make the computer beep a small number of times whenever a window is detected open.

Initially, Heath defined WINDOW as

```
WINDOW;
FOREVER(
   IF EQU? ´ON RECEIVER 3! STATE
      (PRINT [THIEF AT TOP RIGHT WINDOW]; FOREVER(SOUND))
)
```

and the procedure worked very well. He then extended it to detect another window. The modified WINDOW was

```
WINDOW;
FOREVER(
   IF EQU? ´ON RECEIVER 3! STATE
      (PRINT [THIEF AT TOP RIGHT WINDOW]; FOREVER(SOUND))
);
FOREVER(
   IF EQU? ´ON RECEIVER 4! STATE
      (PRINT [THIEF AT BOTTOM RIGHT WINDOW]; FOREVER(SOUND))
)
```

The extended procedure still only worked for the top right window.

Heath made the mistake of not realising that the FOREVER command is non-terminating, so that the second FOREVER command was never executed. He found the program very difficult to debug and gave up. In fact, all he needed to do was to replace the semicolon at the end of the first FOREVER statement by ´//´ so that both commands would be executed in parallel.

Gary´s procedure was

```
WINDOW;
FOREVER(
   IF EQU? ´ON RECEIVER 3! STATE (
      PRINT
        [SOMEONE IS TRYING TO BREAK IN THROUGH THE TOP RIGHT WINDOW];
      FOREVER(SOUND)
   );
   .....
   IF EQU? ´ON RECEIVER 6! STATE (
      PRINT
        [SOMEONE IS TRYING TO BREAK IN THROUGH THE BOTTOM LEFT WINDOW];
      FOREVER(SOUND)
   )
)
```

The limitation of his procedure is that once the alarm was triggered it would continue to beep but would not check whether another window was being broken into.

6.6.2 DOOR

This part of the project was to make the computer be the door keeper. Whenever the door bell was pressed the computer asked for a secret word. If the answer typed in from the keyboard was correct the door would slide open and then close automatically, otherwise the door would remain closed.

Nigel approached the problem in a structured and gradual fashion. He first wrote a procedure that asked for a password and

then checked whether it was correct. He then wrote a procedure that opened and closed the sliding door. Finally he wrote a top level control procedure that detected signals from the door bell. As a result he had a very nice set of nested procedures:

```
DOORBELL;
FOREVER( IF EQU? RECEIVER 7! ´ON (PASSWORD))


PASSWORD;
MAKE ´X ASK [WHAT IS THE PASSWORD];
IF EQU? :X [NIGEL IS GREAT] (DOOR)
   ELSE (PRINT [WRONG. ACCESS IS DENIED TO THE HOUSE])


DOOR;
IF EQU? RECEIVER 1! STATE ´ON (
   @open the door
   SWITCH 2! ON;
   SWITCH 1! ONUNTIL EQU? RECEIVER 2! STATE ´ON;
   @wait for a while
   REPEAT 100 ();
   @close the door
   SWITCH 2! OFF;
   SWITCH 1! ONUNTIL EQU? RECEIVER 1! STATE ´ON
)
```

Michael approached the problem in the same way as Nigel.

Heath first defined a procedure DOOR

```
DOOR;
MAKE ´ANSWER ASK [WHAT IS THE SECRET];
IF EQU? :ANSWER [PIG] (DOOROPEN)
```

which asks for a password and checks it. He also correctly defined a procedure that opens and closes the sliding door:

```
DOOROPEN;
SWITCH 2! ON;
SWITCH 1! ONUNTIL EQU? RECEIVER 2! STATE ´ON;
SWITCH 2! ON;
SWITCH 1! ONUNTIL EQU? RECEIVER 1! STATE ´ON
```

He then completed the problem by extending DOOR so that it checked whether the doorbell had been pressed.

Gary correctly defined the procedure that checked whether the door bell had been pressed, but he had difficulty in defining a procedure to control the sliding door. The author suggested to Gary that he might write two separate procedures, called OPEN and CLOSE. In defining OPEN he made guesses about what the SWITCH commands would be. The author then defined OPEN for him

```
OPEN;
SWITCH 2! ON;
SWITCH 1! ONUNTIL EQU? RECEIVER 2! STATE ´ON
```

and asked him to modify it to make the door close. Gary still ran into all kinds of problems and could not complete the procedure.

6.6.3 House

The final part of this project was to combine the procedures that they had written so far so that the computer could detect thieves coming in through the windows, yet act as doorkeeper at the same time. Nigel quickly recognised the use of parallel processing and defined a procedure

```
HOUSE;
WINDOW // DOORBELL
```

which worked well. The other students tried combining the procedures sequentially but could not get them to work satisfactorily. The multi-programming facility was introduced to all of the students at that point. They greatly appreciated its use.

6.7 Turtle with opto-sensors (Turtle II)

This turtle (see figure 4.2 in chapter 4) differed from the previous one. It was driven by stepping motors instead of DC motors. It also had two reflective-opto switches fixed to its front. A reflective-opto switch sent an ON signal to the computer when it was above a black surface and an OFF signal when it was above a white surface. Figure 6.8 shows the connection of these components.

Component	Concurrent-Logo object
Stepping motor (part 1)	MOTOR 1
(part 2)	MOTOR 2
Reflective-opto switches	
(part 3)	RECEIVER 1
(part 4)	RECEIVER 2

Figure 6.8 Turtle II connection

The objective of the project was to introduce the use of stepping motors and the idea of feedback.

6.7.1 FORWARD, BACKWARD, LEFT and RIGHT

The first task was to write the basic procedures for the turtle. The procedure

```
FORWARD ´X;
MOTOR 1! TURNC :X // MOTOR 2! TURNA :X
```

was given to the students. The procedure takes a number as input. The input value specifies the number of steps that the turtle is to be moved. When the procedure is called, the two commands run in

parallel, causing the motors to turn simultaneously. They were asked to modify it to make BACKWARD, LEFT and RIGHT commands. This task is considerably easier than writing the procedures for the previous turtle; instead of thinking in terms of the SWITCHes operating the DC motors, it is much easier to relate directly the movement of the motors to the movement of the turtle. None of the students had any difficulties in defining the procedures.

The students were also asked to experiment by replacing '//' by ';' in the procedure and observe the effect. Only Michael actually tried it. He excitedly reported to the author that the turtle would rotate right and then rotate left, instead of moving in a straight line. When asked if he understood why, he explained clearly that the motors were not commanded to turn simultaneously.

6.7.2 Set procedure

The next step was to define a procedure that would make the turtle follow a track. The shape of the track is shown in figure 6.1 (Track I).

Heath, Michael and Gary started by writing procedures that directed the turtle to follow the track in a set fashion. The turtle had to start at a fixed point, follow the same path and finish at another fixed point. It was a good exercise for them to estimate angle and distance. It took them some time to get their procedures correct.

Heath had the idea of defining a procedure that would make the turtle move backward to the starting point. He thought all that he had to do was to replace all the FORWARD commands by BACKWARD. The

modified procedure did not work. Heath could not work out what was wrong and gave up. He did not realise that all the RIGHT commands also had to be replaced by LEFT and vice versa. Furthermore, the order of the commands had to be reversed, i.e. the first command of the procedure had to become the last command, the second become the second last, etc.

6.7.3 Responsive procedure

Heath did not attempt this problem.

6.7.3.1 Track I

Nigel and Michael had no problems in defining their procedures that made use of the feedback signals from the reflective-opto switches. Nigel's procedure

```
WALK;
FOREVER(
    @if left side is off the track: turn right
    IF EQU? RECEIVER 1! STATE 'OFF (RIGHT 20);
    @if right side is off the track: turn left
    IF EQU? RECEIVER 2! STATE 'OFF (LEFT 20);
    @if on the track then go forward
    IF ALL EQU? RECEIVER 1! STATE 'ON
           EQU? RECEIVER 2! STATE 'ON (FORWARD 50)
)
```

could make the turtle follow the track both from point A to B and from B to A.

Michael and others took up the author's suggestion of defining two procedures, LEFT.OFF and RIGHT.OFF, for testing whether the turtle's left side or right side was off the track. Michael's WALK was very simple and worked satisfactorily.

```
WALK;
FOREVER(
        FORWARD 20;
        IF LEFT.OFF (RIGHT 20);
        IF RIGHT.OFF (LEFT 20)
)
```

After Gary was persuaded to try he defined WALK as

```
WALK;
FOREVER( FORWARD 100;
         IF LEFT.OFF  (RIGHT 200);
         IF RIGHT.OFF (LEFT  200)
)
```

The algorithm is correct, but the distance moved and the angle turned by the turtle are too great. The turtle would move off the track and be unable to find its way back again.

Gary corrected his procedure in two stages. He first realised that the angle turned was too great, so he reduced the amount of turning from 200 down to 100 and then down to 50. The result of the procedure was still not satisfactory. Then Gary realised the distance moved was too great. He reduced the forward distance to 50 units. His final procedure was

```
WALK;
FOREVER(
        FORWARD 50;
        IF LEFT.OFF  (RIGHT 50);
        IF RIGHT.OFF (LEFT  50)
)
```

6.7.3.2 Track II

After Nigel, Michael and Gary had completed their responsive procedures they were shown another track, figure 6.1 (Track II). Independently they all thought that their procedures could make the

turtle follow the new track. When they tried out their procedures, no matter which end the turtle started from, it would go off the track and be unable to find its way back.

Gary modified his procedure by increasing the units of rotation to 90 degrees. He found that the procedure still did not work and he gave up.

Michael modified his procedure in three steps

(1) by increasing the angle of rotation. He changed the original program to:

```
TURTLE;
FOREVER(
        FORWARD 20;
        IF LEFT.OFF (RIGHT 90);
        IF RIGHT.OFF (LEFT 270)
)
```

and then

```
TURTLE;
FOREVER(
        FORWARD 20;
        IF LEFT.OFF (RIGHT 100);
        IF RIGHT.OFF (LEFT 100)
)
```

(2) to detect whether the turtle was completely off the track. He changed the procedure to:

```
TURTLE;
FOREVER(
        FORWARD 20;
        @if the turtle is off the track: try the left side
        IF ALL? LEFT.OFF RIGHT.OFF (LEFT 90);
        @if still off the track: turn to the right
        IF ALL? LEFT.OFF RIGHT.OFF (RIGHT 180);
        IF RIGHT.OFF (LEFT 20);
        IF LEFT.OFF (RIGHT 20)
)
```

(3) if the turtle is completely off the track to move it back onto the track before making any turns. The procedure was changed to

```
TURTLE;
FOREVER(
        FORWARD 20;
        IF ALL? LEFT.OFF RIGHT.OFF (BACK 20; LEFT 90; FORWARD 20);
        IF ALL? LEFT.OFF RIGHT.OFF (RIGHT 180; FORWARD 20);
        IF RIGHT.OFF (LEFT 20);
        IF LEFT.OFF (RIGHT 20)
)
```

Nigel recognised steps 1 and 2 at once. Only after several modifications to his procedure did he realise step 3 and make the correct modification.

6.7.4 Binary code

The next task was to define procedures for the turtle to recognise a binary-coded pattern. A pattern is represented by four bars. Each bar is either black or white. For example, the patterns shown in figure 6.3 were used to represent the letters ´A´ and ´B´. Fifteen patterns were made up to represent the letters ´A´ to ´O´. The width of each bar is 21 units of turtle movement, i.e. FORWARD 21 would move the turtle from the centre of one bar to the centre of the next one.

6.7.4.1 Recognition I

The author provided the students with a sample procedure

```
CODE.A
@assume the pattern is A
MAKE 'ANSWER 'TRUE;
FORWARD 21;
@if the first bar is not white then false
IF LEFT.ON  (MAKE 'ANSWER 'FALSE);
FORWARD 21;
@if the second bar is not white then false
IF LEFT.ON  (MAKE 'ANSWER 'FALSE);
FORWARD 21;
@if the third bar is not white then false
IF LEFT.ON  (MAKE 'ANSWER 'FALSE);
FORWARD 21;
@if the fourth bar is not black then false
IF LEFT.OFF (MAKE 'ANSWER 'FALSE);
IF :ANSWER (PRINT [THE CODE IS 'A'])
      ELSE (PRINT [THE CODE IS NOT 'A'])
```

which confirms whether a pattern represents the letter 'A'. The procedure assumes that the code does represent 'A' so the first command sets the variable ANSWER to TRUE. It then makes the turtle move from one bar to the next. If it detects that the colour of a bar is not as expected it sets the value of ANSWER to FALSE. If after all the bars have been tested, the value of ANSWER remains TRUE then the code does represent 'A'.

The students were asked to modify the procedure to recognise other letters. Gary did not have time to try. Nigel thought it was too easy and moved ahead to the next part of the project. Michael also found the problem very easy. Just as an exercise he modified CODE.A to recognise the letter 'H'.

Heath did not find the solution obvious. He started by modifying CODE.A to recognise the letter 'B'. He did it in three stages:

(1) He first changed the procedure name to CODE.B and replaced the output lists [THE CODE IS 'A'] and [THE CODE IS NOT 'A'] by the lists [THE CODE IS 'B'] and [THE CODE NOT 'B']

respectively. The result was that it could not recognise the code for ´B´, but it would mistakenly recognise the code for ´A´ as ´B´.

(2) After studying the code more closely, he changed the procedure to

```
CODE.B
@initially set the answer to true
MAKE ´ANSWER ´TRUE;
FORWARD 21;
@if the first bar is white the answer is true
IF LEFT.OFF (MAKE ´ANSWER ´TRUE);
FORWARD 21;
@if the second bar is white the answer is true
IF LEFT.OFF (MAKE ´ANSWER ´TRUE);
FORWARD 21;
@if the third bar is black the answer is true
IF LEFT.ON  (MAKE ´ANSWER ´TRUE);
FORWARD 21;
@if the fourth bar is white the answer is true
IF LEFT.OFF (MAKE ´ANSWER ´TRUE);
IF :ANSWER (PRINT [THE CODE IS ´B´])
      ELSE (PRINT [THE CODE IS NOT ´B´])
```

The procedure recognised every pattern as ´B´, since the first command sets the variable ANSWER to ´TRUE and it is never set to ´FALSE.

(3) After some discussion with the author Heath fully understood the algorithm used in the procedure. He then, by himself, modified the procedure correctly. He also successfully changed it to recognise the letter ´C´.

6.7.4.2 Recognition II

The task was to define a procedure that identified the letter corresponding to a given pattern.

Heath did not try this part of the project.

The only help that Nigel needed was a reminder of how to store values into a list data structure. His procedures were

```
CODE;
MAKE 'PATTERN [];
FORWARD 21;
IF LEFT.ON (MAKE 'PATTERN PUTL 'FALSE :PATTERN)
   ELSE (MAKE 'PATTERN PUTL 'TRUE :PATTERN);
FORWARD 21;
IF LEFT.ON (MAKE 'PATTERN PUTL 'FALSE :PATTERN)
   ELSE (MAKE 'PATTERN PUTL 'TRUE :PATTERN);
FORWARD 21;
IF LEFT.ON (MAKE 'PATTERN PUTL 'FALSE :PATTERN)
   ELSE (MAKE 'PATTERN PUTL 'TRUE :PATTERN);
FORWARD 21;
IF LEFT.ON (MAKE 'PATTERN PUTL 'FALSE :PATTERN)
   ELSE (MAKE 'PATTERN PUTL 'TRUE :PATTERN);
READER

READER;
IF EQU? :PATTERN [TRUE TRUE TRUE  FALSE] (PRINT 'A);
IF EQU? :PATTERN [TRUE TRUE FALSE TRUE]  (PRINT 'B);
IF EQU? :PATTERN [TRUE TRUE FALSE FALSE] (PRINT 'C);
@and so on ....
```

His procedures worked very well. However, they could have been simplified by using a REPEAT loop.

Michael needed much more help. His initial idea was to build a procedure for recognising each letter and then think of a way of combining them. It seemed hard work and he could not think of a way of combining all the procedures even if they were defined.

The author reminded him that the turtle is only expected to walk over the pattern once; it must remember the pattern and then decide what it represents. Once he realised the need to use a variable, the recognition algorithm became apparent. He then wrote a program similar to Nigel's. An interesting variation that Michael tried out was to make the turtle recognise words instead of just characters. He used adhesive to stick together the patterns for the characters

´M´, ´I´, ´C´, ´H´, ´A´, ´E´ and ´L´. To make the turtle walk over the sequence of patterns and the computer spell out his name on the screen he used the command

```
REPEAT 7 (CODE)
```

6.8 Robot arm

The robot arm (see figure 5.3 in chapter 5) has five moving parts: fingers, wrist, forearm, upper arm and shoulder. They are driven by six stepping motors. Figure 6.9 shows the connection.

Component	Concurrent-Logo object
Fingers	MOTOR 1
Wrist	MOTORs 2 & 3
Forearm	MOTOR 4
Upper arm	MOTOR 5
Shoulder	MOTOR 6

Figure 6.9 Robot arm connection

The objective of the project was to introduce different methods of programming a robot, and the ideas of object collision, absolute position and relative position.

6.8.1 Operating the robot arm

Instead of asking the students to write procedures for the arm, they were introduced to a procedure TEACH, written by the author in

Concurrent-Logo, which allowed them to operate the arm by single key-presses. For example, when ´Q´ was pressed on the keyboard the robot opened its fingers, and when the key ´W´ was pressed the robot raised its wrist. Alltogether twelve keys were used (see Worksheet 14 in Appendix I for more detail).

The students learned the TEACH procedure very easily. However, it took them some practice before they could operate the arm smoothly. The common tasks that they did with the arm were

(1) pick up a block and put it into a box

(2) pick up a lid and put it on the box

(3) unstack a tower of blocks.

6.8.2 Teaching the robot arm

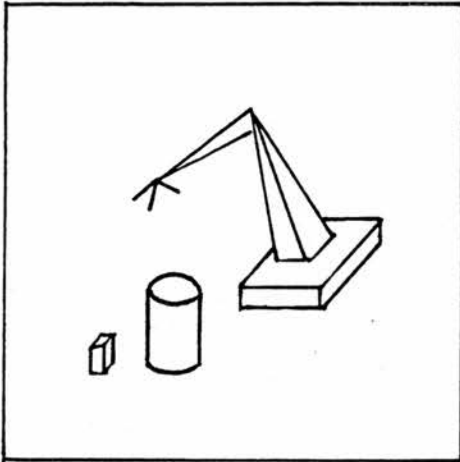
When the students were familiar with TEACH and with operating the arm, the author introduced three more procedures: REMEMBER which appended the present position of the arm, represented by a list of six numbers, to a list called SEQUENCE; REPLAY which moved the arm to its start-up position then moved the arm through the sequence of positions as recorded in SEQUENCE, and FORGET which set SEQUENCE to a null list.

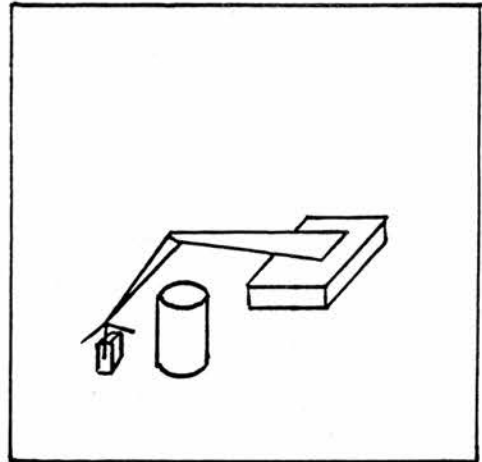
The students were asked to repeat the tasks listed in the previous section. For each task, at any stage, they could request the computer to remember the arm position. After they had operated the arm to accomplish the task, they could ask the computer to replay the arm movement by making it follow the sequence of recorded positions.

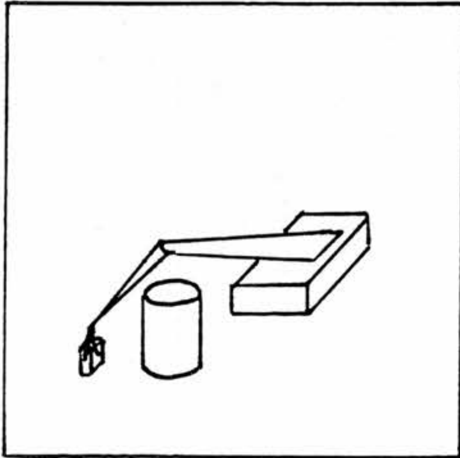
None of the students could make the arm replay a sequence of actions successfully the first time. They all made one characteristic mistake: for each task, certain essential arm positions were not recorded.

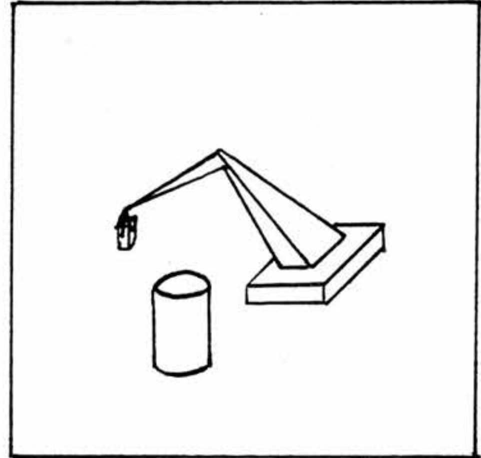
Task 1

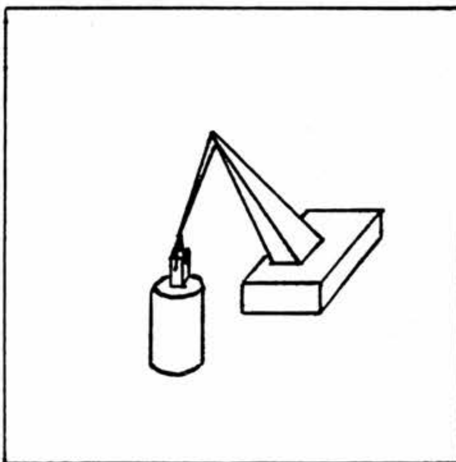
Figure 6.10 shows the essential arm positions that must be recorded.


Position 1


Position 2


Position 3


Position 4


Position 5

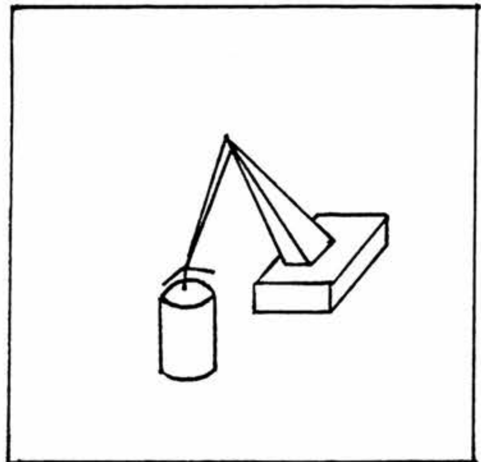

Position 6

Figure 6.10 Sequence of robot arm positions (II)

All of the students recognised position 1 and position 3 as important and recorded them. The positions that the students omitted were 2 and 4.

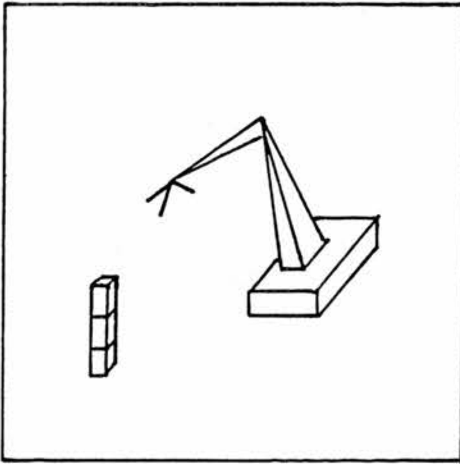
The replay showed up the mistakes clearly. As the arm moved from the start-up position directly to position 3 its fingers were closing at the same time. When the arm reached the block its fingers were closed already and the block was knocked over. With position 4 omitted the arm took a direct path from position 4 to position 5 and on its way it also knocked the box over.

Task 2

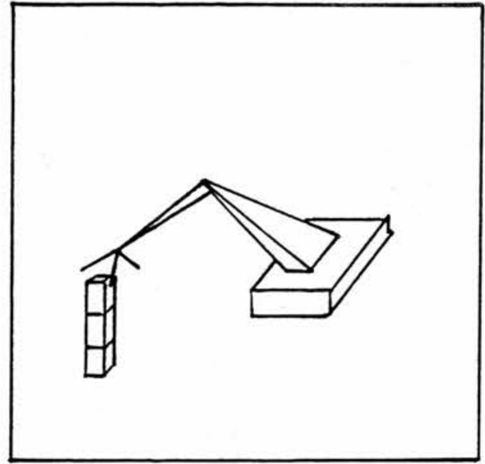
The mistakes that the students made in task 2 were similar to that of task 1.

Task 3

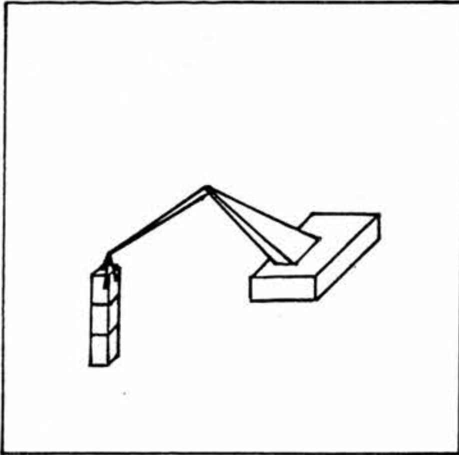
Figure 6.11 shows the positions that had to be remembered to unstack the top block.



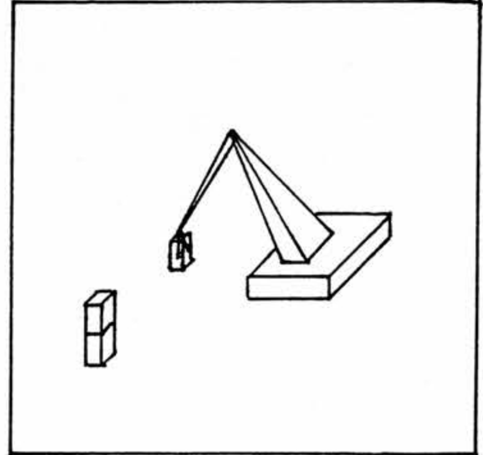
Position 1



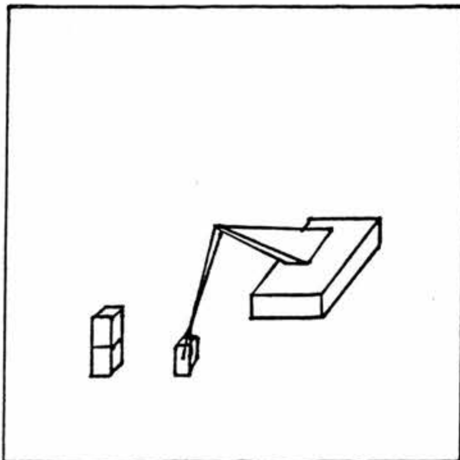
Position 2



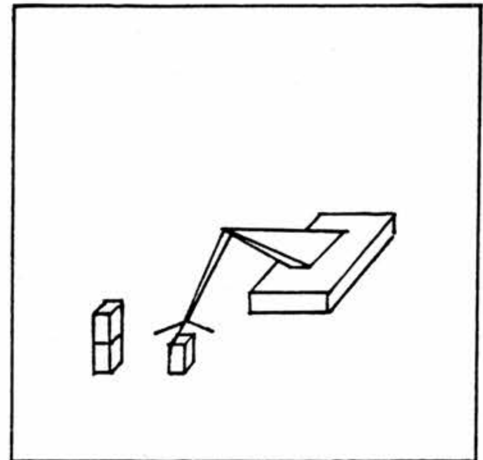
Position 3



Position 4



Position 5



Position 6

Figure 6.11 Sequence of robot arm positions (III)

Heath and Gary continued to make the mistake of omitting position 2. The position that all of the students omitted was 4. The consequence was that the arm took the direct path from position 3 to 5 and it made the remaining tower fall over.

The students had a lot of fun playing with the robot arm. It also gave a very good opportunity for the author to discuss with the students the effect of wrongly programmed robots in a real working environment where the damage could be costly and men's lives endangered.

6.8.3 Programming the robot arm

The students were taught how to program the arm using MOTOR commands directly.

Two worksheets were used. One explained the relationship between the arm's movement and the MOTOR commands; the other described two procedures, POSITION and MOVETO, that they might use for developing their own programs. POSITION returned the current position of the arm as a list of six numbers. MOVETO took a list of six numbers as input and moved the arm to the position as defined by the list.

The students had no difficulty in relating the MOTOR commands to the movement of the arm. When they had typed a MOTOR command they could see the corresponding arm movement. However, Heath and Gary had difficulties in understanding the number representation of arm position.

Nigel successfully defined two procedures: BLUE and IN.BOX. BLUE moved the arm to an absolute position where a blue block was

placed and then gripped it. IN.BOX put whatever was in the arm's fingers into a box. After he had defined these two procedures he typed the command

```
BLUE; IN.BOX
```

which made the arm pick up a blue block and then put it into a box. His procedures were:

```
BLUE;
@move the fingers close to the block
MOVETO [-205 161 72 55 -225 61];
@get it exactly right
MOVETO [-205 161 72 55 -295 51];
@close the fingers to grip
MOVEMOTOR 1 269;
@raise the shoulder
MOVEMOTOR 5 -95;

IN.BOX;
@move the arm so that it is above the box
MOVETO [269 161 72 -83 99 -181];
@open the fingers and let whatever is in it drop into the box
MOVEMOTOR 1 69;
```

To define these procedures Nigel used TEACH to operate the arm. Once it was at the right place he used POSITION to find out its position.

MOVEMOTOR was defined by the author. Its first input was a number specifying a MOTOR; it moved that MOTOR to the position specified by the second input.

6.9 Discussion

The preceding description shows the variation of the students' work, especially the difficulties that they had and the ways that they solved them. However, the practical problems of teaching a course that involves lots of equipment do not come across clearly from the description. The rest of this section is in two parts. The

first identifies the benefits and limitations of learning control applications through programming. The second discusses the practical problems of safety and convenient arrangement of equipment.

Later chapters describe the analysis of the students' understanding, students' response and the language.

6.9.1 Benefits and limitations

One clear advantage of programming is that it compels a student to think about how something is done. For a student to think about and describe an algorithm has much more educational value than for him to learn a set of rules or a sequence of operations without understanding why they work. Programming controllable devices is particularly suited for learning because the program's operations are externalised through the device's actions. These actions provide information that can be interpreted against the student's intentions for the program. He can identify mismatches between the expected behaviour and the actual behaviour of the device. However, this does not mean that he can correct the program easily. The mismatches could be caused by ill-conceived algorithm or faulty implementation of the program. The latter type of mistakes could be trivial, like typing errors or mis-spellings. It could also be at the conceptual level, like misunderstanding the semantics of certain control structures. Thus correcting a program, besides revising the description, means that the student has to improve his understanding of the problem and/or the programming language. From the students' work there are good examples illustrating that correcting programs is a constructive learning process.

A distinction should be made between using direct commands and writing programs. Sometimes a student can manipulate a device in a step by step fashion but has difficulties in specifying the algorithm, or in implementing it as a program, or both. One example is to make the lift move to different floors. The students can easily do that using direct commands. However, writing a program requires a student to identify explicitly the information that he uses and the decisions that he makes instinctively. In this particular example the information is the current position of the lift and the decision is the direction in which the lift should move. Another example is to make a turtle follow a track. Driving the turtle directly, a student might be unaware that he makes use of information about the current state of the turtle to decide whether the turtle should go forward, turn left or turn right. Writing the program WALK (section 6.7.3) forces a student to represent the knowledge explicitly. The difference between using direct commands and writing programs is similar to that between operating the robot arm and teaching it. As described in section 6.8, the errors that the students made in teaching the arm revealed the kind of information that they had overlooked. Direct control is less demanding because a student does not have to plan in detail. At this stage, the knowledge could be vague. Programming helps a student to bring this knowledge to the surface by making it explicit in programs. This also leads to the conclusion that the demonstration approach as described in section 2.4 is very limited, because a student is not led directly to think how a device is controlled. Even if different control algorithms were used to control a device, a student would not be able to perceive that just from watching a demonstration. However, he

would have gained the appreciation that a device can be programmed to do certain functions.

Learning through programming provides an excellent context for teacher-student interaction. A student's programs reveal his difficulties and misconceptions. A teacher is more able to give help that meets the individual's needs. Furthermore, the teacher-student discussion centres on correcting the computer and not the student.

After a student has had some experience with a model, the teacher can help him to relate the model to real life applications. The information that is passed on then becomes much more accessible to a student as he has a concrete example and experience that he can reference.

The programming approach has its intrinsic values. However, teacher-student interaction is still fundamentally important. The teacher has to understand the students' problems; he has to give careful guidance to lead students from one level of understanding to another and he has to grasp the opportunities to engage students in purposeful discussion.

In this study the students used some very powerful programming constructs including procedures, list processing, various looping structures and parallel processing. However, the students did not use these constructs in depth. Typically, they wrote procedures that were only two to three levels deep. Their use of parallel processing was also limited to one or two simple applications, e.g. Doll's house and Turtle II. The students used only objects that were provided in the Concurrent-Logo systems and did not define their own. If the aim

of the course were to be extended from giving students experience in using these constructs in constrained situations to teaching them how to apply these constructs to a wider range of tasks then the number of projects would have to be expanded and more worksheets devised.

6.9.2 Equipment arrangement

At the practical level, the teacher also has to consider how equipment is to be arranged in the classroom. In the study two computers were used in each session. Initially each sub-group worked on the same project and one device was shared between the two computers. This arrangement was found to be very inconvenient. While one sub-group was testing their program the other group had to wait. Another problem was that the device had to be frequently unplugged from one computer and plugged into the other. To avoid moving the device every time, the computers had to be placed very close to each other with the device between them. It was very distracting. To remedy the problem two different devices were introduced at once so that each computer had a device dedicated to it. In a real classroom where there are more students and more computers, equipment arrangment deserves careful attention.

The issue of safety should not be ignored when mobile devices are directly under the students' control. In particular students do write incorrect programs and cause the devices to be out of control. One example is programming the lift. A student wanted to make the lift move up and stop at the third floor but the stopping condition was wrong, so the motor carried on turning even after the cage had passed the third floor. When the cage had reached the top of the frame and could not move any further the string that pulls the cage

became tighter and tighter. The author had to quickly switch off the power supply to the motor to prevent the string from snapping. The sliding door of the doll's house, the robot arm and even the turtle can be potentially dangerous. The most important thing is that the power supply switches must be within easy reach and that every student must know where they are.

CHAPTER 7

STUDENT'S UNDERSTANDING

At the beginning of the course it was difficult to assess the students' understanding of control applications since they had no formal training in this subject and there are no standardised tests. Therefore, the students were not given a pre-test. Based on the author's observation and the content of the course taught to them, by the end of the course the students were expected to have gained

(1) appreciaion of the different applications they had tried out;

(2) knowledge of the electronic components used;

(3) understanding of the concept of feedback;

(4) knowledge of the basic problems and limitations in the point to point control of robots;

(5) an appreciation of the use of procedures;

(6) an appreciation of the power of parallel processing.

A test (Appendix III) was designed and used to examine the students' understanding in the latter five areas. Since there is only a post-test and no pre-test, the result cannot be used to suggest how much each student had improved through taking the course. Its contribution is that it identifies whether certain concepts can be learned easily. The result also provides guidelines for designing future tests.

The students were not told that they had to sit a test so they did not purposely prepare for it. One student, Heather, thought when she read the questions that they were too hard and asked not to answer them. Since her help was voluntary the author agreed. Therefore, the assessment is based on the written answers of eleven

students. They tended to give short answers expressing their primary and partial understandings. In the following sections, quotations of students' answers are not altered and any underlining is the author's emphasis.

7.1 Components

Two types of question were asked about components. The first type was factual. The second type required more understanding and judgement to answer.

The factual questions were:

(1) What is a push-button for?

(2) What is a reed switch for?

(3) What is a reflective-opto switch for?

All the students answered the factual questions fairly well. The experience with the devices provided a link for the students to relate to the components and what they were used for.

One of the more difficult questions was 'What are the differences between working with DC motors and stepping motors?' Since the students were not taught electronics they were only expected to point out the functional difference. Seven students answered satisfactorily. A sample answer:

> 'Stepping motors only turn the number of steps they are told while DC motors turn continuously'

The observation, though trivial, was missed by four of the students.

The final two questions asked the students to choose the electronic components for two computer controlled toy models: a cable car and a crane.

Knowledge of the difference between controlling DC motors and stepping motors is necessary to answer the toy model questions correctly. For the cable car, it was to move cyclically from the bottom to the top and then from the top to the bottom. It is better to use a DC motor to drive the car and use reed switches to test whether the car has reached either the top or the bottom. For the crane, the moving parts were to move a specified amount at a time. It is better to use stepping motors.

Five students had given the expected answer to both questions. These students not only knew about the components but also appreciated their applications and limitations.

One student preferred using DC motors for the crane and explained that repeat loops could be used to control the amount moved. Most students had chosen to use stepping motors for both of the models.

7.2 Feedback

The feedback concept was not taught explicitly. However, it was applied in the projects that made use of sensors, namely, Lift, Doll's house and Turtle II. When asked the question: 'What are the advantages of having sensors attached to a control device?', ten students appeared to be aware that feedback helps the computer to know something is going wrong. A typical answer is:

'It (the computer) can detect if it (the device) is going off course.'

Two of them, Nigel and Kevin, added the point that feedback helps the computer to make decisions. A specimen answer is:

'The advantage of having sensors on a control device is that the computer can be helped to make up it's mind what to do.'

Michael also made another point: feedback helps the computer to take corrective actions. His answer was:

'The computer then know if the device has gone off course and take steps to correct it.'

Although the concept of feedback

feedback information -> make decision -> take corrective actions

could be learned through the programming activities alone, notice that the use of the phrase 'going (or gone) off course' in the first and third answers above shows that the students were actually referring to a particular experience, i.e. programming the turtle with sensors to follow a black line. This project had given them the deepest impression of the use of feedback.

7.3 Robot

The students were asked the question: 'What are the difficulties in training a robot arm to do a sequence of actions?' None of the students gave a complete answer. They tended to describe only one or two aspects of the difficulties. However, a collection of their answers would constitute a complete answer. Here are some quotations from the students:

'You have to get it to remember each step'

'You have to make it remember positions in the right place'

'It has to remember a large number of actions in the right order'

'Making sure the arm doesn't take short-cuts knocking over the object it is supposed to pick up'

'It tends to take the direct route from point A to B and not the route you want it to take'

'If the position of something is changed then it will not work'

These quotations came from six students' answers. They describe fully the sequence and the limitations, using the point to point control method. These ideas were learned naturally from the programming activities. However, each student had a deeper impression of a particular aspect and was more able to recall it.

7.4 Procedures

The use of procedures is fundamental to the learning through programming approach. Teachers write procedures to demonstrate concepts which students can explore and use to solve problems. Students use procedures to solve problems in a structured and gradual fashion.

To find out how well the students had understood the use of procedures, they were asked two questions:

(1) Do you think procedures are useful? If so, why?

(2) If you know BASIC, can you tell me what the differences are between subroutines in BASIC and procedures in Concurrent-Logo?

The author was surprised to find that only nine, not all, of the students thought that procedures were useful. The two who

considered procedures were not useful gave the reasons, 'BASIC is easier to edit' and 'BASIC is much easier to edit'. These comments will be considered later. The students' understanding of procedures can again be categorised into three levels:

(1) procedures as editing tools

(2) procedures as problem solving tools

(3) procedures as language extension tools

The first level is concerned predominantly with text manipulation, i.e. how to enter and modify a program text. The second level is appreciating that the use of procedures makes programming easier. The third level is understanding that a procedure, once defined, can be used just as if it was a primitive, thus extending the programming language.

The two students who did not like procedures obviously belong to the first level. They overlooked the purpose of procedures and concentrated on the editing facilities. Their view of programming was distorted by their experiences with BASIC. Instead of seeing a program as a collection of procedures they saw a program as a long sequence of instructions. Towards the end of the second term, one of them asked, 'Why can't we write long programs like in BASIC. We have only been writing short procedures.'

The rest of the students had reached at least level two understanding. Two of the students expressed their ideas quite clearly:

'They (procedures) make the program less messy, easier to write and easier to read. It is more structured and logical.'

'Procedures are useful as they let you see bugs in the program

more easily.'

However, only two students, both of whom understood BASIC, showed level three understanding, i.e. procedures as language extension. One of the students wrote 'The main difference between subroutines and procedures is that subroutines need a lot of looping and will not normally work on their own.' He was implicitly saying, 'BASIC subroutine requires the use of GOSUB and RETURN statements; procedure is called by name. Subroutines can only be used inside a program; procedures, once defined, can be used on their own.' Other students gave no indication that they had understood the extensibility of Concurrent-Logo. Students who knew BASIC could only point out the superficial difference between BASIC subroutines and Concurrent-Logo procedures: line numbers are used in the former and not in the latter.

It is interesting to note that the best two and the worst two answers were all given by students who knew BASIC. Thus experience with BASIC might or might not have a deleterious effect in moulding the students' attitude towards programming.

7.5 Parallelism

One of the questions that the students were asked was: 'Do you consider parallelism is an important part of a programming language? Why?'. Keith abstained and the rest gave a positive answer to the first part of the question. Different reasons were given as answer to the second part. Five of them gave answers similar to:

'it is important to be able to do a few things at once'

Ruth made the same point by referring to one particular device: 'in the robot arm you really have to make the arm go down and closing the fingers at the same time'. Its ability to move different parts at the same time obviously made an impression on her. Kevin mentioned that 'it makes things clearer to understand.' Gary explained that parallellism makes the language more flexible.

Two of the students identified parallelism as important because of faster execution speed. This, however, is not strictly true because Concurrent-Logo provides only pseudo-parallelism, i.e. time sharing. Of course, they were not aware of that.

The students were also asked questions related to understanding the flow of control. One question asked them to describe the effects of the following commands:

```
a) REPEAT 10 (PRINT 'A); REPEAT 10 (PRINT 'B)
b) REPEAT 10 (PRINT 'A) // REPEAT 10 (PRINT 'B)
c) FOREVER (PRINT 'A); FOREVER (PRINT 'B)
d) FOREVER (PRINT 'A) // FOREVER (PRINT 'B)
```

Nine of the students described a) and b) correctly. An example is:

a) it will print 10 A's then 10 B's.
b) it will print 10 A's and 10 B's simultaneously.

One of the students gave a totally wrong answer. He wrote:

a) print A and B.
b) print A and B twice.

Another student gave the same description for both of the commands: 'print A and B 10 times.' Her description is ambiguous; it makes no distinction between commands that are to be executed in sequence or in parallel. From this example alone it is difficult to know whether

she was just being vague or had a misconception. The descriptions of c) show that not only she but also some of the others did not fully uderstand the flow of control of programs. An example of a wrong answer is:

c) it will print A and B until you tell it to stop.

Understanding that the FOREVER loop is non-terminating is easy. The difficulty lies in realising that because the two FOREVER commands are in sequence the second one would never be executed. Their model seems to be that when the computer executes a FOREVER command it does what it has been told to do continuously but also carries on finding out what else it has to do. If the computer is told to do two things together it will literally do them simultaneously. Their model is closely related to human behaviour. For example, it make sense to ask a person to sing and dance at the same time. On the other hand a person can be told to sing continuously and while he is doing that he is also able to respond to further commands such as 'also dance continuously'. However, there is ambiguity in the latter way of instructing a person to do things. Does it mean 'sing a bit and dance a bit continuously' or 'sing and dance at the same time continuously'. Depending on the situation, a person might be able to resolve this kind of ambiguity. The students obviously expected the computer be able to do that as well.

Only four students could answer correctly all the questions related to control flow of programs. Two of them were the most experienced programmers and the other two were the most able students of the third year group.

An example of their descriptions of c) and d) is:

c) it would just print A forever until you tell it to stop.
d) it would print A and B forever at the same time.

7.6 Conclusion

The overall result of the test is tabulated in figure 7.1, and individual student's results are tabulated in figure 7.2. It is evident that a basic understanding and appreciation of control technology can be learned through programming. However, in general, it is difficult to gain profound understanding from the activities alone. It could be that the students were not very good at structuring their ideas and describing them. Supplementing the programming activities with formal discussion and some teaching might help to improve their understanding.

On the whole, the fourth year students had done better than the third year students, especially on the question about DC motors and stepping motors. The questions on the flow of control of concurrent programs are particularly difficult. The limitation of the computational model has to be explained to them clearly.

	Types of component questions		
	factual	motors	model
No. of correct answers	11	7	5
No. of incorrect or unanswered	0	4	6

	unanswered	Levels		
		1	2	3
No. of student at different level of understanding of procedures	1	2	6	2

	number of points mentioned about robot control	
	1	2
number of students	9	2

	number of points mentioned about feedback		
	0	1	2
number of students	1	7	3

	Types of multi-programming question	
	appreciation	flow of control
No. of correct answers	10	4
No. of incorrect or unanswered	1	7

Figure 7.1 Overall result

Notes to figure 7.2:

(1) The students' results are tabulated in the order of overall achievement.

(2) The small letters besides each student's name are the teachers' grading of the students learning ability.

(a) well above average

(b) above average

(c) just above average

(d) average

(e) just below average

(f) below average

(g) well below average

(3) Keys to the tables

'/' means correct answer

'X' means wrong answer

'-' means did not answer

Fourth year group:

	Component Question			no. of points		level	parallelism	
	factual	motor	model	feed-back	robot	pro-cedure	apprecia-tion	flow of control
Nigel (b)	/	/	/	2	2	2	/	/
William(c/d)	/	/	/	1	1	3	/	/
Heath (e)	/	/	/	1	1	2	/	X
Martin (d)	/	/	/	1	1	2	/	X
Neil (b)	/	/	X	1	1	2	/	X

Third year group:

	Component Question			no. of points		level	parallelism	
	factual	motor	model	feed-back	robot	pro-cedure	apprecia-tion	flow of control
Michael(a)	/	/	/	2	1	2	/	/
Keith (a)	/	/	X	-	2	-	-	/
Kevin (b)	/	X	X	2	1	3	/	X
Ruth (e)	/	X	X	1	1	2	/	X
Lynette(b)	/	X	X	1	1	1	/	X
Gary (d)	/	X	X	1	1	1	/	X
Heather(d)	-	-	-	-	-	-	-	-

Figure 7.2 Summary of students' results

CHAPTER 8

STUDENT'S RESPONSE

Since the teaching of control applications is still at the pioneering stage, it is important to find out any information that would help the development of future work. A survey was designed, by the author, to find out what the students thought about the course after they had completed it.

On the whole, the students had enjoyed the course and felt that they had benefited from it. None had developed a dislike of control applications. The students' preferences for devices were varied, confirming the view that a course in control applications should cover a wide range of devices rather than just use one device to teach different concepts.

This chapter ends with a discussion on how the course might be improved.

8.1 Course

All twelve particpants started the course wanting to know more about computing, a result obtained from the first survey (Questionnaire 1 in Appendix II). Ten of them maintained their interest throughout the whole course. The students' answer to the question 'Did you find the course enjoyable?', given in the final survey (Questionnaire 4 in Appendix II), is shown in figure 8.1.

Most enjoyable	3
Enjoyable	7
Fair	2
Boring	0
Very boring	0

Figure 8.1 Students' opinion of the course

Neil and Gary were the two who found the course fair. Neil's problem was not that he didn't enjoy learning control applications. He just did not get on with his partner. His resentment is obvious in his answer to the question 'How many people do you prefer to work with?': he preferred to work by himself. All other students preferred to work with either one or two friends.

Gary had an inflated assessment of his ability. Although he showed a poor understanding of his work, he always thought that the projects were easy. Hence, he was not fully immersed in the activities.

It is important to note that Neil and Gary were not the least able students. Their personal difficulties can only be dealt with successfully by skilful teaching.

The majority of the students found the notes helpful and clear (see figure 8.2). Half of them would have liked to receive more notes. All felt that they had received sufficient help when they needed it. Seven stated that they would have preferred more teaching

and explanation about control applications.

very helpful	3
helpful	6
fair	3
not helpful	0
not helpful at all	0

very clear	2
clear	7
fair	3
not clear	0
not clear at all	0

Figure 8.2 Students' opinion of the worksheets

In response to the question 'Do you think you have learned anything useful?', all the students, except Neil, gave a positive reply, figure 8.3.

a lot	2
quite a lot	4
some	5
a little	1
very little	0

Figure 8.3 Students' own evaluation of how much they had benefited

These answers indicate that the students saw some educational value in the course and they felt they had achieved something through it.

Again, with the exception of Neil, the students gave positive answers (figure 8.4) to the question, 'Would you recommend the course to your friend?'.

yes	11
no	0
not sure	1

Figure 8.4 Students' recommendation of the course

Eight of the students requested to continue in the next academic year.

8.2 Projects

Although the students responded positively, it is necessary to know more about what the students enjoyed and why, in order to refine and to extend the course.

Figure 8.5 shows the students' preferences among the control devices.

Project / preference	1st	2nd	3rd	4th	5th	6th
Windmill	0	0	1	1	2	8
Turtle	0	2	2	3	5	0
Lift	1	1	3	2	3	2
Doll's house	0	2	5	3	2	0
Turtle with sensors (Turtle II)	2	5	1	2	0	2
Robot arm	9	2	0	1	0	0

Figure 8.5 Students' rating of the devices

As the survey shows, the robot arm was the most popular and the windmill was the least. There is also a trend, but less apparent, in the second, third, fourth and fifth choices, that is Turtle II, Doll's house, Lift, and Turtle respectively. In fact the trend is set by the more able students. They had all included Robot arm and Turtle II in the top two preferences and mostly with Doll's house and Lift as their third and fourth. These are interesting devices that the able students could do, and had done, a lot with. It is not surprising that the able students preferred them. The average or less able students, though many of them had put the robot arm as their first choice, were more diverse in their other choices. For example, Gary had chosen Turtle, Lift and Windmill as his second to fourth preferences respectively; Heather had chosen Doll's house, Windmill and Turtle II; Heath had chosen Turtle, Doll's house and Turtle II. The evidence shows that, to an average or less able student, a simple windmill or turtle could be a better learning

device than the more complicated ones. It is interesting to note how Gary had struggled with the Turtle project and yet it is his second choice.

The three most common reasons that the students gave for choosing their three favourites are:

(1) practical

(2) can do a lot with it

(3) can understand it.

These reasons are certainly true for the robot arm. It is practical and versatile. The students found it easy to use because they were given a set of predefined procedures to operate it. Otherwise many students would have had difficulties.

Some of their specific programming experiences also have influence on their choices. Heath explained that he liked the Doll's house because 'you could combine all your programs to work at once'. Lynette wrote: 'I enjoyed the turtle with the opto sensors the most, as it was good for trying to write a program to keep it on the black line. In the end, we (she and her partner) succeeded. This was a happy moment for us all.'

Certain characteristics of the devices also appealed to individuals: Heather liked incorporating sound in the Doll's house project; Lynette liked the way that the lift went up and down; William liked the Robot arm because he had seen the same type of Robot arm shown on a television computer program.

The students' opinions about whether they had spent enough time on each of the projects are quite mixed. Their answers to the

question, 'Would you like to have spent more, or less, time with the devices?', are tabulated in figure 8.6.

device / time	more	about right	less
Windmill	2	7	3
Turtle	6	5	1
Doll's house	6	4	2
Lift	5	4	3
Turtle with sensors	6	5	1
Robot arm	8	4	0

Figure 8.6 Students' opinion of amount of time spent on each project

8.3 Discussion

From the motivational point of view, the course appeared to be a success. The students, of varying abilities, had developed, or maintained, an interest in control applications. There was no feeling that 'only the clever people can do control applications' or 'only the boys are good at it'. One possible reason for this success was that they perceived programming control devices as fun, so they enjoyed it. Another possibility might be the Hawthorne effect. The students had been chosen to take part in an experiment and they might have put in extra effort to make the experiment successful. If the latter is the case, in the long run one would expect to detect a drop in the motivational level. However, the study's duration was too short for this to be investigated.

The students' preference of devices confirms the principles we used in designing a device:

(1) it should have the potential for doing interesting things

(2) more importantly, the interesting things should be within the students' capabilities.

Furthermore, to meet the needs of individual students a collection of devices should be used, rather than just one.

The strategy of introducing simple devices first and leading on to more sophisticated ones, as used in this study, could be profitable. The important factor is to give the students enough time. When a new device is introduced, a student should have the choice of carrying on working with the previous one, if so wished. There is great value in encouraging a student to persevere with a project to the end, as shown in reasons given by some the students on why they liked a particular device. A student should not be rushed from one device to the next, otherwise what he could gain is minimal and superficial. Only when he has grappled with a project for a sufficiently long time does the learning become personal and rewarding.

Unfortunately, in this study some students felt that they were being rushed. A fixed amount of time was allocated to each project. When a new project was introduced the students were not given the choice whether they could carry on working with the previous one. This decision was taken to make classroom management easier. It minimised the problem of moving and connecting different devices to the computers during one session and the author could concentrate on helping and observing the students.

Ideally, the students should be allowed to work at their own pace. However, it remains important that their activities be guided by structure. Throughout the study the author was often asked by the students the question, 'What should we do next?' When they had finished a program it was difficult for them to decide on something else to do because designing a task for a control device requires some appreciation of what the device can do in the first place. It is very different from deciding what pattern to draw next with a drawing device. Good suggestions must be planned beforehand so the students may be initiated into thinking about complex control algorithms.

CHAPTER 9

ASSESSING CONCURRENT-LOGO

This chapter first reviews some of the work previously done in programming language design and then assesses Concurrent-Logo in the light of the previous work and the experience gained in the pilot study.

9.1 Programming language design

9.1.1 An overview

There are three stages in programming language design:

(1) identify the general requirements of the language

(2) specify and implement the language

(3) evaluate the language

The general requirements of a programming language for teaching and learning at secondary level were described in chapter 1. It must be interactive, extensible, visible and simple. It is worth noting that this set of requirements is different from those commonly found in programming language design text books (for example see Horowitz, 1983; Young, 1982). These books deal mainly with software development languages. Therefore, they would include other requirements such as security, efficiency and portability. At this early stage of identifying the general requirements, the language designer must have a clear idea of who the intended users of the language are and how it is intended to be used.

The common features found in most high level languages are: block structures, control structures, data structures, arithmetic

operations, assignment operations and I/O operations. However, languages can be very different in their syntax and details (for example see Sammet, 1969; Tennent 1981). Hoare's (1973) advice is that

> the language designer should be familiar with many alternative features designed by others, and should have excellent judgement in choosing the best and rejecting any that are mutually inconsistent. He must be capable of reconciling, by good engineering design, any remaining minor inconsistencies or overlaps between separately designed features. He must have a clear idea of the scope and purpose and range of application of his new language, and how far it should go in size and complexity.

Most language features can be implemented using established techniques (Aho and Ullman, 1978; Brown, 1979). If a programming language includes novel facilities that are difficult to implement efficiently, then a considerable amount of research effort has to go into designing implementation techniques. One example is Ada's tasking facilities for communicating sequential processes that run in parallel (Habermann and Nassi, 1980). Another example is Prolog. Since it uses unification and backtracking as its basic execution model, new compilation techniques and new ways of representing the internal data structures have had to be devised (Mellish, 1982; Warren, 1983; Clocksin, 1985).

The third stage of the design process is to evaluate the language. There are two aspects of language evaluation. One is from the technical point of view. It is concerned with evaluating the speed, storage usage and reliability of particular implementations. The other aspect is from the users' point of view. It is concerned with identifying oddities, ambiguities and missing facilities of a programming lanugage. This chapter considers this latter aspect of

evaluation in respect of certain details of syntax and facilities.

9.1.2 Language evaluation

9.1.2.1 Syntax

The syntax of a programming language can have a significant effect on the readability of programs. The principle that Wirth (1974) gave was

> The language should not be burdened with syntactical rules, it must be supported by them. They must therefore be purposeful, and prohibit the construction of ambiguities.

Though the principle is simple, its application is subjective. In practice, the real issue seems to be finding a compromise between clarity, convenience and flexibility.

Ripley and Druseikis (1978) carried out an analysis of the syntax errors of 589 Pascal programs written by students. They found that 41% of the errors were omitting a single syntactic token in a statement, about half of which were the statement separator ';'. There were 83 instances of missing ';', significantly all of which, except one, occured at the end of a line. The conclusion is that there is a strong tendency for programmers to regard the end of a line as the end of a statement. It may seem that line-oriented languages are to be preferred. However, they create other problems. Take Logo for example. One common complaint is that there is no way of writing a program with indentation to reflect the structure of a program or statement (Hardy and Hardy, 1985).

For example the following Logo REPEAT statement

```
REPEAT 6
  [
   REPEAT 60
     [
      FORWARD 1
      RIGHT 1
     ]
   REPEAT 60
     [
      FORWARD 1
      LEFT 2
     ]
  ]
```

may appear on a 40 column screen as

```
REPEAT 6 [ REPEAT 60 [ FORWARD 1 RIGHT !
1 ] REPEAT 60 [FORWARD 1 LEFT 2 ] ]
```

The absence of a command separator also creates an ambiguity when multiple commands are on the same line. For example, the line

```
PROC_1 PROC_2
```

can mean

(1) there is one command PROC_1, which takes one argument the result returned to it by PROC_2; or

(2) there are two commands PROC_1 followed by PROC_2.

The Logo interpreter resolves this kind of ambiguity at run time by checking the number of arguments each procedure has when it is called. However, to make programs more readable, 380Z Logo (Johnson, 1983) introduced the word 'and' for separating commands on the same line. Unfortunately the introduction of this new syntactic token received much criticism. It is made optional in a later version of Logo from the same designers and implementors. There is certainly a

conflict between the demands of clarity and those of convenience.

Another example of compromising clarity for the sake of convenience is the use of infix operators in Logo. For example, the commands

```
PRINT FIRST 234          would print 2
PRINT 234 + 2            would print 236
```

but what is the effect of

```
PRINT FIRST 234 + 2
```

Is it

```
PRINT (FIRST 234) + 2    which would print 4
```

or is it

```
PRINT FIRST (234 + 2)    which would print 2
```

Cannara (1975) recommended infix operators should be left out of Logo to avoid this kind of ambiguity. It was a deliberate decision that infix operators was left out of 380Z Logo (Ross and Howe, 1984). However, by popular demand, they were introduced into the later version.

Arblaster (1982) mentioned that a common error in using Pascal is mismatching the 'begin's and 'end's that mark the start and the end of a block of statements respectively. In Ripley and Druseikis' study cited above, mismatching 'begin's and 'end's accounted for 8% of the total errors. The additional rule of requiring a full stop after the final 'end' in Pascal also caused problems. The suggested solution is to have more explicit bracket pairs like if - endif, for

- endfor, while - endwhile. COMAL does this. The solution in Ada is similar. Instead of using a concatenated word as a closing bracket, it uses two existing reserved words in sequence, for example, if - end if, loop - end loop. These solutions improve clarity at the expense of making the syntax of the language more complicated. Lisp is open to the same kind of criticism. The only bracket pair it uses is '(' and ')'. A solution which does not involve changing the syntax is to provide editors that have syntax checking capabilites. One such capability is to make the cursor jump to the matching parenthesis (for example the 'emacs' editor (Stallman, 1985)).

In Logo a word may be prefixed by either of two special characters: a word preceded by a single quote (or double quote in some implementations) means a constant or a name of a variable; a word preceded by a colon means the value of the variable; with no prefix, a word means a procedure invocation. The following are all valid Logo commands

```
MAKE 'X 'Y        MAKE :X 'Y        MAKE X 'Y
MAKE 'X :Y        MAKE :X :Y        MAKE X :Y
MAKE 'X  Y        MAKE :X  Y        MAKE X  Y
```

which have totally different effects. The syntax is extremely powerful but difficult to use, especially for novices (du Boulay, 1978). They have difficulty in distinguishing the difference between a name, the value of a variable and the value returned by a function. Most programming languages avoid this problem by dereferencing from name to value automatically. For example, the variable 'I' appears twice in the Pascal statement

```
I := I + 1;
```

The first 'I' stands for the name (reference) and the second stands for the value. Automatic dereferencing is done at the cost of increasing the difficulty of using higher order variables. The following Logo code can increment any named variable:

```
INC 'VAR_NAME
MAKE :VAR_NAME ADD 1 VALUE :VAR_NAME
```

In Pascal, the user has to learn about the difference between 'pass by value' and 'pass by reference', and the syntax of declaring different kinds of variable.

One final point about syntax is that command names should be short and meaningful. This would reduce some common errors that students make:

(1) mistyping

(2) confusing the meaning of different words. An example, drawn from an early implementation of Logo in Edinburgh, is the use of the words REMEMBER and RECALL for storing procedures into the filing system, and retrieving them from it, respectively. Some students were confused because they thought of the words as synonyms (Ross and Howe, 1984)

(3) misusing space characters. Commands that are made up of concatenated words, such as 'PENUP', mislead students into thinking that the space delimiter is unimportant. As a result they would type commands like 'FORWARD100' or even 'PEN UP'.

9.1.2.2 Facilities

When a programming language is put into practical use, it is inevitable that the users will find something that they would like to

do but which the language does not allow them to do easily. The criticisms about the facilities of a programming language can be divided into three groups:

(1) new routines need to be added to the library utilities

(2) existing facilities need to be respecified

(3) extensions need to be added to the language

Obvious examples of the first group are formatted read and sorting procedures. Where these procedures can be implemented in the language itself, it is a good design principle to leave them out of the core definition. The advantage is that the language would be small, including only the essentials, which makes learning easy. A prerequisite is that the language must be extensible, providing facilities for linking and loading pre-compiled utility routines. For interactive languages like Logo, the utility routines are usually in the source form. It would be a good idea if new implementations allow utility routines to be compiled and then dynamically loaded in at run time.

If existing facilities need to be respecified it is usually due to the oversight of the designer. For example, some complaints about Pascal are that variable declarations do not allow initialization and that the 'case' statement does not have an 'otherwise' clause for specifying default actions (Mickel, 1981)

Other limitations of a programming language can be dealt with only by extending it. In particular most programming languages provide only one level of modularity, namely procedure, and do not support concurrent programming. This had led Brinch Hansen and Wirth to extend Pascal to Concurrent Pascal and Modula respectively. An

important point is that the extension should at the same abstraction level as the rest of the language (Wirth, 1974).

Language designers should be concerned not only about the facilities of the language but also facilities for debugging programs. In particular the error messages that the system generates should be at a level of detail appropriate to the programmer's understanding of the computational events. Error messages from compilers are useful for correcting syntactic errors. However, run time error messages of compiled languages are notoriously bad. It is no use to the programmer if the system generats an error message like

Segmentation violation

and then aborts. A language implementation, whether the language is compiled or interactive, should provide a debugger so that a programmer can trace and follow the execution of his program.

9.1.3 Summary

Designing a programming language is a complex process. More importantly the process is a co-operative one. Hoare's (1973) advice is

> Listen carefully to what language users say they want, until you have an understanding of what they really want. Then find some way of achieving the latter at a small fraction of the cost of the former. This is the test of success in language design, and of progress in programming methodology.

Some compromises that designers have to make about the syntax of a language have been described, and criticisms concerning the facilities of a programming language were divided into three groups. Different action should be taken depending on the nature of the

criticism. Finally, a language implementation should be supported by debugging aids.

9.2 Concurrent-Logo

Most of the criticisms of Logo have already been illustrated in the previous section. They can be summarised as

(1) Logo does not allow textual layout of a program to reflect its structure

(2) some of the command names are not carefully chosen

(3) the use of a quote and a colon to distinguish between the name and the value of a variable often cause problems for novice programmers

These criticisms are concerned with syntactical issues. Recently, Hardy and Hardy (1985) also discussed the need to extend the facilities of Logo.

The rest of this chapter concentrates on evaluating the extended facilities found in Concurrent-Logo; features of the language that are likely to cause programming errors are identified; the limitation and further extension of the language are also discussed.

9.2.1 Cause of errors

Figure 9.1 shows which of the extended commands or facilities were used during the pilot study.

Extended command or facility	Used by students	Used by author in demonstration programs
SWITCH n ! ON	/	/
SWITCH n ! ONUNTIL	/	/
RECEIVER n ! STATE	/	/
RECEIVER n ! KEEPCOUNT		
RECEIVER n ! COUNT		
MOTOR n ! TURNC	/	/
MOTOR n ! TURNA	/	/
MOTOR n ! COUNT	/	/
MOTOR n ! STOP		/
//	/	/
FOREVER	/	/
GUARD		/
USER DEFINED OBJECTS		/
WHENEVER		/

Figure 9.1 Facilities used in Concurrent-Logo

When the students used the extended facilities there were three main sources of error:

(1) the extra syntax markers: semicolon, parallel bars and exclamation mark;

(2) the RECEIVER object and command;

(3) the FOREVER command.

The post-test given to the students also showed that they had difficulty in understanding the flow of control of programs.

9.2.1.1 Syntax

With the extended facilities, two major syntax rules were introduced in Concurrent-Logo:

(1) Semicolon and parallel bars '//' were used as command separators: if two commands were separated by a semicolon they would be executed in sequence; if two commands were separated by parallel bars they would be executed in parallel. One or the other was obligatory.

(2) Exclamation mark was used to separate an object from its message.

In practice the need to type in any syntax markers can be a source of error and a cause of frustration. The students remembered the exclamation mark after one session and took over three sessions to get used to the semicolon. By the time the parallel bar notation was introduced the students were quite well acquainted with the idea of a separator, so they learned the new notation with relative ease.

For multi-programming it is necessary to have a syntax marker to indicate processes that are to be executed in parallel. The parallel bar symbol is as good as, if not better than, any other symbols since it is simple and has the right connotation. Therefore, it should not be changed.

The semicolon was introduced for two reasons. One is consistency - all commands are separated by markers. The other reason is to allow free-form layout of programs. It improves program

clarity but is a hindrance to the users. An alternative solution, which allows free-form layout of programs and does not require any explicit command separator, is to treat 'carriage returns' or 'linefeeds' in a procedure body as space characters. The effect is that the procedure body is interpreted as if it is all typed in on one line. An additional advantage of this solution is that it is backwards compatible with existing implementations of line-oriented Logo. It is worth experimenting with this solution and see whether it creates any problems for users.

The object metaphor was very good for explaining the RECEIVER, SWITCH and MOTOR commands. However, students could easily leave out the exclamation mark between an object and its message. The object metaphor could be discarded by changing the RECEIVER, SWITCH and MOTOR commands to conventional procedure form. Figure 9.2 shows the equivalence of the two forms. If the object approach is discarded, for the sake of consistency, the synchronization mechanism will also have to be redesigned.

Object form	Procedure form
RECEIVER n ! STATE	R.STATE n
RECEIVER n ! KEEPCOUNT	KEEPCOUNT n
RECEIVER n ! COUNT	R.COUNT n
SWITCH n ! ON	ON n
SWITCH n ! OFF	OFF n
SWITCH n ! STATE	S.STATE n
SWITCH n ! ONUNTIL condition	no equivalent
MOTOR n ! TURNC m	TURNC n m
MOTOR n ! TURNA m	TURNC n m
MOTOR n ! COUNT	M.COUNT n
MOTOR n ! STOP	M.STOP n

Figure 9.2 Commands in object and in procedure forms

Note that the object form allows different objects to have the same message name. For example, both RECEIVERs and SWITCHes can receive the message STATE. Also, both RECEIVERs and MOTORs can receive the message COUNT. The object name provides the context for the message. In procedure form, no two primitives or procedures can have the same name. Therefore, some of the primitives need to have suffixes.

To find out what the students preferred, they were asked three questions in the final questionnaire:

(1) To make stepping MOTOR 1 turn clockwise 300 steps, which command do you prefer (or suggest your own)?

a) MOTOR 1! TURNC 300

b) MOTOR 1 TURNC 300

c) TURNC 1 300

(2) To turn SWITCH 2 ON, which command format do you prefer (or suggest your own)?

a) SWITCH 2! ON

b) SWITCH 2 ON

c) ON 2

3) To find out the state of RECEIVER 3, which command format do you prefer (or suggest your own)?

a) RECEIVER 3! STATE

b) RECEIVER 3 STATE

c) STATE 3

Note, all of the b) options are syntactically incorrect, because the names of the objects would be interpreted as procedure calls. The options were included to find out whether the students preferred including the object name in a command.

The students' answers are shown in figure 9.3.

Question / preference	a	b	c
1	6	5	1
2	6	2	4
3	6	2	4

Figure 9.3 Students' preference for formats

The author stressed to the students that there were no right or wrong answers to these questions.

The answers shows that the majority favoured naming both object and message (options a) and b)). The use of the exclamation mark is accepted by half the students. Since the students answered the questionnaire after they had learned Concurrent-Logo their answers may be biased towards what was familiar. Comparing only options b) and c), the students prefer syntax c) if a command has at most one input. Otherwise b) is preferred.

9.2.1.2 RECEIVER

The word 'RECEIVER' is difficult. The students often mis-spelled it. To avoid distraction a simpler word should be used. A better word may be 'SENSOR'.

The STATE command caused some confusion. It was used most in the condition part of an IF command. For example

```
IF EQU? RECEIVER 1! STATE 'ON  (action 1)
IF EQU? RECEIVER 2! STATE 'OFF (action 2)
```

The STATE command is objectionable because

(1) the code for testing the state of a RECEIVER is very long

(2) the students often thought, mistakenly, that STATE returned either the word 'TRUE or 'FALSE.

To overcome the deficiences it would be better to replace STATE by two commands ON? and OFF?. The command RECEIVER n ! ON? would return 'TRUE if RECEIVER n were on and 'FALSE otherwise. The command RECEIVER n ! OFF? would return 'TRUE if RECEIVER n were off, and

'FALSE otherwise.

The examples above would become

IF RECEIVER 1! ON? (action 1)

IF RECEIVER 2! OFF? (action 2)

It would be even simpler if the word RECEIVER is replaced by SENSOR.

9.2.1.3 FOREVER command

The FOREVER command is simple both syntactically and semantically. It is a loop structure without a stopping condition. In its simplest form the students had no difficulty in understanding or using it. However, there was a common mistake that the students made: putting two FOREVER commands in sequence. A simple example would be:

FOREVER (action 1); FOREVER (action 2)

When a mistake of this type is made, the intended format of the command is either

FOREVER (action 1; action 2)

or

FOREVER (action 1) // FOREVER (action 2)

These two formats have very different effects. The first would execute action 1 and action 2 alternately forever. The second would execute the actions forever in parallel.

The problem with the FOREVER command is not that its syntax or semantics is obscure but that the students have difficulties in

understanding the interaction between the command and the part of the program that it appears in. They need to be taught how the FOREVER command should be used. This finding is in accord with the conclusion drawn by Soloway and Ehrlich from their study (1982). They emphasised the importance of teaching more than just the syntax and semantics of looping structures. The context of their study is quite different from the present one. They tested whether student programmers could distinguish the appropriate context in which to use each of the Pascal's looping structures (the FOR, REPEAT and WHILE loops). They found that the students (even after taking a course in Pascal programming) had chosen the correct structure less than half the time. The result demonstrates that the students should be taught when and how the structures should be used.

9.2.2 Limitation and extension

The prototype implementation of Concurrent-Logo has two major deficiencies:

(1) the programming environment is primitive;

(2) the language is inadequate for applications that involve collecting data at a fast rate or in a large quantity.

The inadequacy of the programming environment is identified below, and extensions for data collection facilities are suggested.

9.2.2.1 Programming environment

A particular advantage of an interactive language is that it has an integrated programming environment. It should include an editor, filer and debugger. In the prototype implementation, due to the limited memory capacity of the TERAK computer, all these facilities

are either very basic or non-existent.

The screen editor is very primitive. It has no copy, search or replace capabilities. As a user grows more sophisticated the absence of such facilities can be most annoying.

Commands are available only for saving, retrieving and erasing the definitions of procedures, guards and messages. There is no way of saving the entire content of the working memory. Once the computer is switched off, all the global variables and their values are lost. Therefore, at the beginning of a session a user might have to spend some time re-initialising the system to its previous state.

There is virtually no debugging facility. When an error occurs an error number is printed instead of an error message. A user then has to refer to a list of error messages.

All the above mentioned shortcomings can be successfully dealt with if the next version of Concurrent-Logo is developed on a computer that has more memory capacity than the TERAK.

A less trivial problem is that of providing suitable tracing facilities. Since there may be more than one active process, multiple window displays have to be used to show the progress of the different processes.

9.2.2.2 Data collection

An example of an application that requires collecting data at a fast rate and in a large quantity is the control of chemistry and physics experiments. In this type of application the source data are usually in analogue form and need to be converted into digital form

for the computer. The digital representation of an analogue signal is usually a number in the range 0 to 255.

To conform with the object approach, a solution would be to introduce a class of system objects called PORT. Different PORTs are identified by numbers, i.e. PORT 1, PORT 2 and so on. Each PORT would respond to four different messages:

(1) VALUE

E.g. PORT 1 ! VALUE

which tells PORT 1 to return the value of the present signal.

(2) COLLECT n m

(n is the number of signals to be collected and m is the delay time (in centi-seconds) between reading the signals)

E.g. PORT 2 ! COLLECT 1000 50

which tells PORT 2 to collect 1000 signals with half a second delay between each reading.

(3) ITEM n

E.g. PORT 2 ! ITEM 250

which tells PORT 2 to return the value of the 250th collected item.

(4) RECORD

E.g. PORT 2 ! RECORD

which tells PORT 2 to return all the recorded values as a list.

These PORT instructions allow convenient analogue input and rapid data collection.

9.3 Conclusion

The prototype version of Concurrent-Logo forms the basis of a practical control language for educational use. However, for it to be fully useful some improvement is needed. The areas that need particular attention are the programming environment and the extension for data collection. The improved version of Concurrent-Logo should be sufficiently general for most educational control projects. Future development should be done on a 16 bit computer, with at least 128K random access memory. For maximum run-time speed the interpreter should be written in assembler language. If portability is important then an implementation language should be used, but it is essential that a compiler exist for compiling the source code directly into machine code.

An instruction method also has to be developed to help students to understand the control flow of programs.

CHAPTER 10

CONCLUSION

10.1 Summary

The research described in this thesis investigated the teaching of computer control applications at upper secondary school level. The review showed that manufacturers have produced useful control devices for educational use. One significant advancement is the development of general purpose control modules which enable control devices to be built very easily and cheaply by teachers or students using model construction kits. The major problem that is identified as hindering the teaching of control applications is the lack of a suitable computer language and courseware. Consequently a course in control applications is biased towards teaching of electronics or demonstrating what a control device can do. Neither of these emphases helps students to focus on thinking about the control process of a system. A related problem is that there is little knowledge of how students learn control applications. In order to improve the teaching it is necessary to know what kind of misconceptions students hold and what common mistakes they make.

The first contribution of this research is the design of Concurrent-Logo. It is a computer language based on Logo, a language well developed for educational purposes but lacking facilities for control applications. The extensions included in Concurrent-Logo are commands for detecting signals, commands for actuating switches and stepping motors, multi-programming, and control structures for handling events.

A course in control applications was also developed. It consists of six projects. Each project involves writing programs for a particular control device. The control devices are: windmill, turtle, doll's house, lift, turtle with optical sensors and robot arm. These devices make use of a wide range of electronic components and the programming tasks also cover a wide range of control and computing concepts.

A prototype implementation of Concurrent-Logo and the course were tested in a pilot study involving twelve students. The study demonstrated that it is practicable to learn control applications through programming with a high level computer language. A student very often can manipulate a device in a direct drive mode. However, he may be unaware that he is instinctively making use of feedback information such as the current state of the device. Programming helps him to bring knowledge to the surface by making it explicit in programs.

A profile of four students' work is given. An analysis revealed several misconceptions that the students had and the mistakes that they made. Examples of problematic areas are controlling DC motors, pulsing, pattern recognition and teaching a robot arm.

The post-test results showed that the students had grasped some of the control concepts required in the applications. However, they did not show in-depth understanding. This could be due to three factors. First, a student was not given enough time to pursue a project for as long as he wanted. Second, the author wanted the students to learn as much as possible from their own programming activities and did not fully explain the concepts to them. Third,

the testing material was not sensitive enough to find out all that a student had learned.

The final survey showed that the students' response was positive. They enjoyed the course and felt that they had learned something from it. The ideas and materials developed for this course could be used by curriculum developers as the basis for designing a practical course.

Certain deficiences of Concurrent-Logo have been identified. The object name RECEIVER is difficult to spell. The RECEIVER command STATE is confusing. Facilities for fast data collection are required. Suggestions for improvements are given.

10.2 Criticism

There exist an implementation of Concurrent-Logo and worksheets for the course. However, the implementation is only a prototype and the worksheets were developed specifically for the pilot study. Both of these need to be improved before they could be used widely in schools.

The pilot study was carried out with a small number of students. The author played multiple roles in the study as tutor, as experimenter and as evaluator. From a methodological standpoint, this situation was not ideal. However, it had to be so because of limited resources.

With hindsight, it is clear that the final survey might have been administered by a school teacher instead of by the author, to make it less obvious that the survey was part of the experiment.

That would have reduced the possibility that the students gave favourable answers just to please the author.

In the absence of standardised tests for control applications, the author had to construct his own test in an ad hoc fashion. His task was the harder because very little is known about students' understanding in this subject. The post-test was constructed in an effort to verify the author's expectation that the students would have gained some appreciation or understanding of the electronic components used, the concept of feedback, the basic problems and limitations in the point to point control of robots, the use of procedures, and parallel processing. The test was not sensitive enough to reveal all that a student had learned. Though the results indicated that the students had difficulty in understanding the control flow of parallel programs, the collected data were not sufficient to infer students' mental models. The test procedure might have been improved by interviewing each student as well. With appropriate prompting during an interview, a student might have revealed more than he wrote on paper.

From the experimental design point of view, it would have been better if a pre-test had been given. Then, an analysis of the pre-test and post-test results would show how much each student had benefited from the course. Based on the experience gained from the study, a new pre-test and a new post-test are devised, which are included in appendix IV.

10.3 Future research

This thesis provides some preliminary evidence to support the proposed innovation. One direction of research is to go through a second phase of monitoring stage evaluation. It should be carried out after the refinement of software and hardware. More reliable and sensitive tests should also be produced. If the outcome continues to be favourable, a larger scale study may then be justified. The number of classes should be increased. The number of students in each class should be about the same as in a normal class. Furthermore, the teaching should be done by teachers other than the committed investigators. Such a study would provide stronger evidence of the practicability of the approach. At the same time, it is necessary to continue collecting detailed data that would help to improve our knowledge of students' understanding of control applications.

It has been mentioned that students have difficulties in understanding the control flow of concurrent programs. An instructional method needs to be developed to help students to understand parallel processing. A tentative suggestion is the idea of 'playing computer'. For example, one student can act as a computer that obeys command sequentially. When he encounters commands that are to be executed in parallel he will organise them and delegate them to his fellow students, one command for each of them. When all his fellow students have finished what they have been told to do he will continue to obey the next command. Different models of parallelism can be tried out in this way. The concept of mutual exclusion can be illustrated clearly if the students were asked to act out a command like:

boil water with the red sauce pan //
warm up the baked beans with the red sauce pan

Another direction of research is integration. It is mentioned in the first chapter that control application is related to many subjects. It is worthwhile investigating these relationships. The aim is to design learning activities that will bring multi-disciplinary skills and knowledge into one context. For example, after the students have had sufficient experience in programming control devices, they may design their own devices. This provides the students with practical work that would help them to appreciate physics concepts such as equilibrium, mechanical advantage, efficiency and velocity ratio. The students can also exercise their judgement in selecting the appropriate electronic components. If sufficiently motivated, they may also design and assemble the electronic modules required.

Finally the transition from secondary education to higher education should also be considered. The approach proposed in this study is on a practical basis. The teaching of control theory and robotics at university level is undoubtedly on a theoretical basis. The missing links that would help students to progress naturally from concrete to abstract and from practical to theoretical need to be identified.

APPENDIX I

IMPLEMENTATION OF CONCURRENT-LOGO

This appendix is in four sections. The first gives an overview of the major components of the Concurrent-Logo system. The second describes how data is held in the main memory. The third describes the run time local stacks and how concurrency is implemented. The last section gives the syntax of Concurrent-Logo as implemented.

I.1 The software

The prototype system was implemented on the Terak 8510. The implementation language was UCSD Pascal, which is a structured high level language well suited for software development. The UCSD Pascal system also has the extended facility for overlay; where the code and data for a segment procedure are in memory only while there is an active invocation of that procedure. However, a snag is that a UCSD Pascal program is not directly compiled into machine executable code. The program is compiled into an intermediate form called P-code and the P-code program is interpreted at run time, which reduces the performance. To compensate the deficiency, time critical routines were written in assembler and linked together with the compiled Pascal code.

The Concurrent-Logo system has four main components: a parser, an interpreter, an editor and a filer.

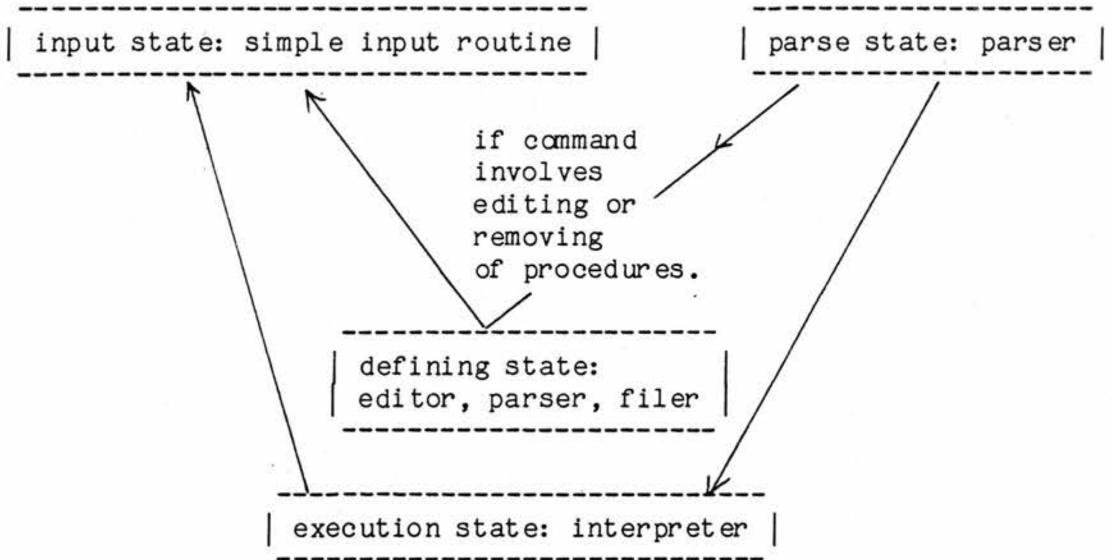
The parser is responsible for parsing any input text making sure that it is syntactically correct and also for translating it into an internal format suitable for execution by the interpreter. The internal format of a command is in postfix form and is stored as a list. Examples are given in section I.2.

The interpreter is responsible for the execution of instructions in the internal format.

The filer is responsible for saving and retrieving the source of procedure definitions onto and from the disk respectively. This simply involves keeping a directory of where a procedure is stored on disk and initiating block transfers between the main memory and the disk.

The built-in editor is screen based. It provides the basic facilities for moving the cursor around the screen and for insertion and deletion of text.

The interaction between these components is illustrated below.



I.2 Format of internal data structures

Beside a text buffer for temporarily holding user input and text for editing, the Concurrent-Logo system has eight main data areas:

(1) A word space, where all previously unrecognised text is stored if necessary.
(2) A variable name space, for holding the information associated with names of variables.
(3) A procedure name space, for holding the information associated with names of procedures.
(4) A class name space, for holding the information associated with names of classes.
(5) An object name space, for holding the information associated with names of objects.
(6) A guard name space, for holding the information associated with names of guards.
(7) A list space, for keeping track of lists, procedure definitions, guard definitions and class procedure defintions.
(8) Eight local stacks, for keeping track of control information, procedure parameters and local variables while executing commands.

Most of these areas are divided into records. Free records within each of these areas are chained together via their last fields. To reduce the implementation effort, a garbage collector was not implemented. Instead, when used records become free they are put back on the appropriate free list immediately. The following describes (1) to (7) separately and then gives snapshots of the internal memory illustrating how these areas relate to one another. The description of the local stacks is in section I.3 where the run time organisation of the system is described.

I.2.1 The word space

This space is initialised to hold the names of all the system procedures and objects. As the system runs, any new text strings, such as procedure names and variable names not previously recognised by the system are inserted. Component strings in this space are stored consecutively without any separators. There are no pointers from this space to anywhere. A pointer into this space is usually stored together with a number indicating the length of the string allowing text to be used as identification tags or for printing purposes.

I.2.2 The variable name space

This space contains information about variables. The space is divided into records and each record has six fields:

POINTER TO VARIABLE NAME (in the word space)
LENGTH OF NAME
TYPE OF VALUE
VALUE
LENGTH
POINTER TO NEXT RECORD

If TYPE OF VALUE is

(1) number, VALUE holds the value of the number as a 16-bit signed integer, and LENGTH is not used.
(2) word, VALUE holds the starting address of the word to be found in the word space, LENGTH holds the length of the word.
(3) list, VALUE holds the starting address of the list to be found in the list space, LENGTH holds the number of elements in the list.

I.2.3 The list space

This space contains information about lists and code for procedures, guards and objects. The space is divided into blocks and each block has four fields. In a list, a record looks like this

TYPE OF VALUE
VALUE
LENGTH
POINTER TO NEXT RECORD

If TYPE OF VALUE is

(1) number, VALUE holds the value of the number as a 16-bit signed integer, and LENGTH is not used.
(2) word, VALUE holds the starting address of the word to be found in the word space, LENGTH holds the length of the word.
(3) list, VALUE holds the starting address of the list to be found in the list space, LENGTH holds the number of elements in the list.

For storing procedure code, the first field of a record is the opcode and it determines how the second and third fields are used.

I.2.4 The procedure name space

This space contains information about user-defined procedures. The space is divided into records and each record has five fields:

POINTER TO PROCEDURE NAME (in the word space)
LENGTH OF NAME
POINTER TO NAMES OF FORMAL PARAMETERS (in the variable name space)
POINTER TO PROCEDURE CODE (in the list space)
POINTER TO NEXT RECORD

I.2.5 The guard name space

This space contains information about user-defined guards. The space is divided into records and each record has five fields:

POINTER TO GUARD NAME (in the word space)
LENGTH OF NAME
POINTER TO GUARD CODE (in the list space)
PROCESS NUMBER
POINTER TO NEXT RECORD

The PROCESS NUMBER of a guard is initialised to 0. When the guard is told to wakeup a new process is allocated for the running of the guard code. The new process is identified by an integer between 1 and 8 and this value is stored in PROCESS NUMBER. So, when the guard is told to sleep the corresponding process can be identified and removed.

I.2.6 The class name space

This space contains information about different classes of objects. The space is divided into records and each record has five fields:

POINTER TO NAME OF CLASS
LENGTH OF THE NAME
POINTER TO CLASS PROCEDURES (in the procedure name space)
POINTER TO NAMES OF OWN VARIABLES (in the variable name space)
POINTER TO NEXT RECORD

I.2.7 The object name space

This space contains information about user-defined objects. The space is divided into records and each record has eight fields:

POINTER TO NAME OF OBJECT (in the word space)
LENGTH OF THE NAME
POINTER TO CLASS INFORMATION (in the class name space)
POINTER TO OWN VARIABLES (in the variable name space)
OBJECT STATUS
POINTER TO WAIT QUEUE
POINTER TO DELAY QUEUE
POINTER TO NEXT RECORD

When a new object is created a new object name block is allocated for it. POINTER TO CLASS INFORMATION points to the class name block of which the object is an instance. New variable name blocks are allocated for the object's own variable.

The fields OBJECT STATUS, WAIT QUEUE and DELAY QUEUE are for the implementation mutual exclusion and synchronisation. To recap, the special features of an object are:

(1) if an object receives more than one message simultaneously, they will be processed strictly one at a time.
(2) while obeying a message, if the object encounters the DELAYIF statement and the condition is evaluated to TRUE, the request is delayed, and the object makes itself available to other requests. After a request has been successfully processed the object will check whether it can restart any previously delayed requests.

Initially OBJECT STATUS is set to 0 and both the WAIT and DELAY QUEUEs are empty. This means that the object is passive - no message is sent to the object to run any of the procedures defined for the class it belongs to. When a message is sent to the object the OBJECT STATUS is set to 1. During the execution of the procedure if another process sends a message to the same object that process is suspended and put on the WAIT QUEUE. The first process in the WAIT QUEUE will be reactivated when the object has finished the execution of its current procedure and has checked the DELAY QUEUE.

I.2.8 Snapshots of the internal memory

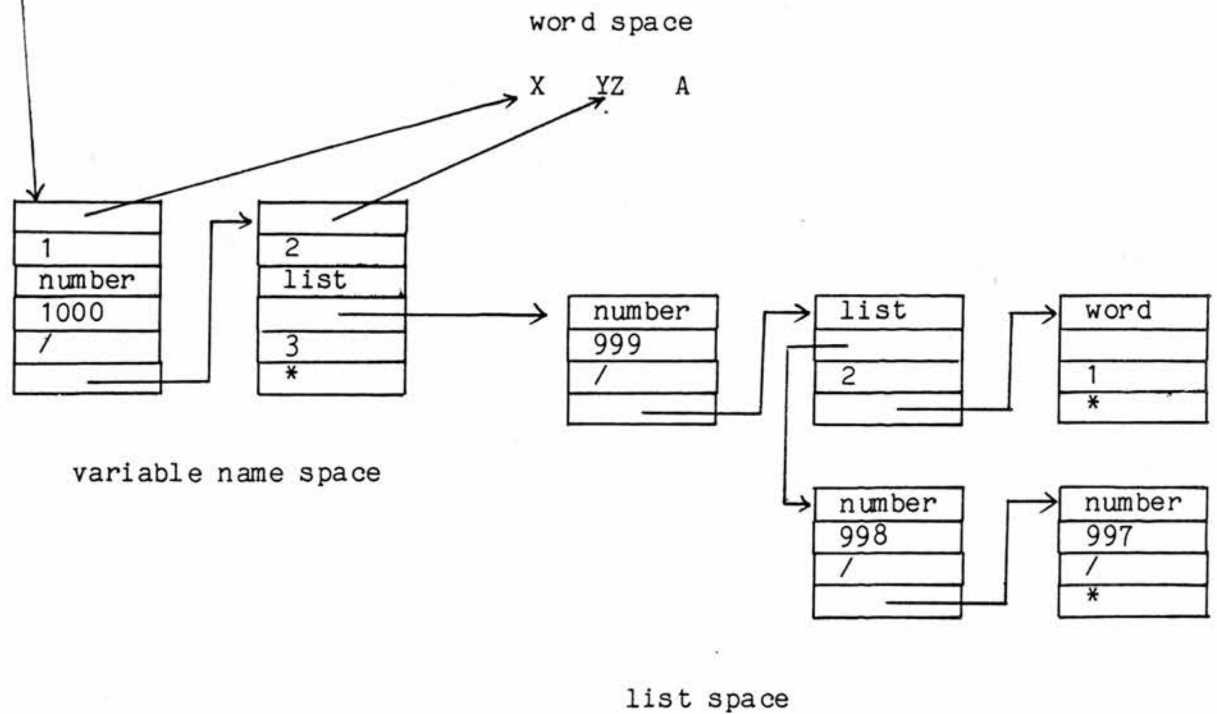
The following are snapshots of part of the interal memory. They illustrate more clearly how the internal memory is arranged and how procedures definitions are stored. In the illustrations, the end-of-list character is '*'.

Snapshot 1

After the execution of the commands

MAKE 'X 1000; MAKE 'YZ [999 [998 997] A]

part of the memory looks like:



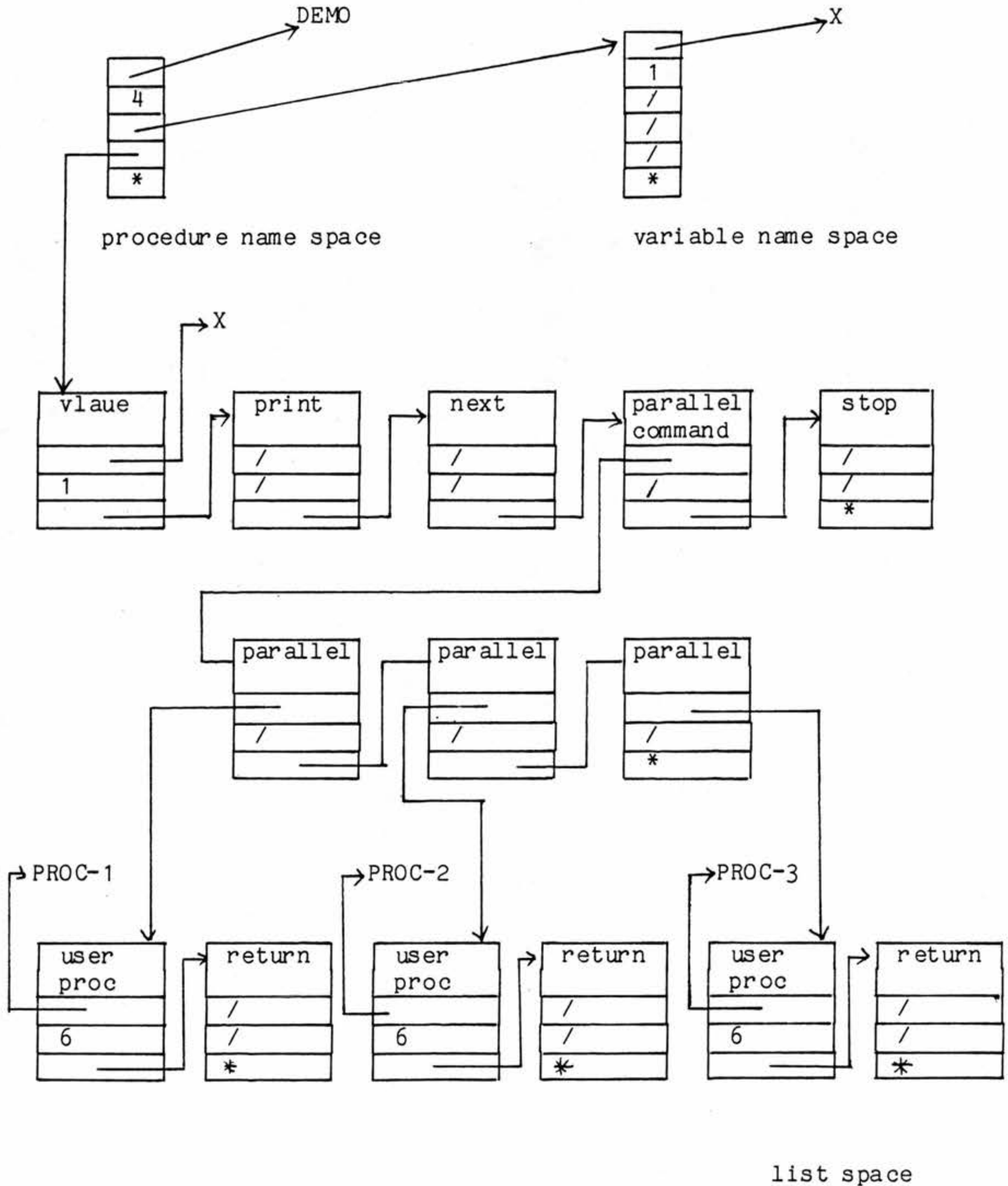
Snapshot 2

If a procedure DEMO is defined as

```
DEMO 'X;
PRINT:X;
PROC-1 // PROC-2 // PROC-3
```

part of the memory looks like:

procedure pointer



Snapshot 3

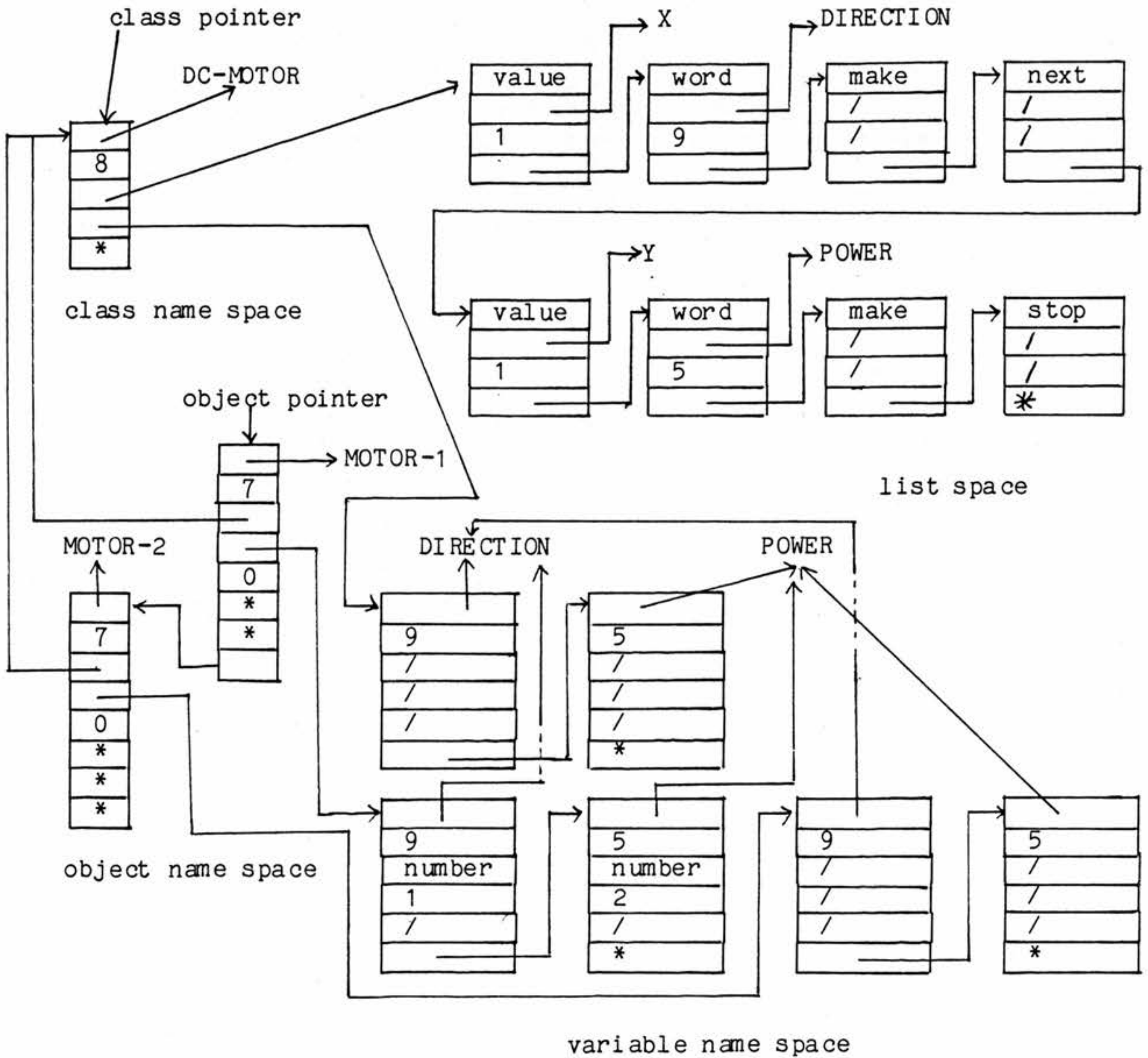
After the execution of the commands

```
NEWCLASS 'DC-MOTOR HAS 'DIRECTION 'POWER
NEWOBJECT 'MOTOR-1 CLASS 'DC-MOTOR
NEWOBJECT 'MOTOR-2 CLASS 'DC-MOTOR
DEFINE 'INIT CLASS 'DC-MOTOR
MOTOR-1 ! INIT 1 2
```

where INIT is defined as

```
INIT 'X 'Y
MAKE 'DIRECTION :X
MAKE 'POWER :Y
```

part of the internal memory looks like:



Snapshot 4

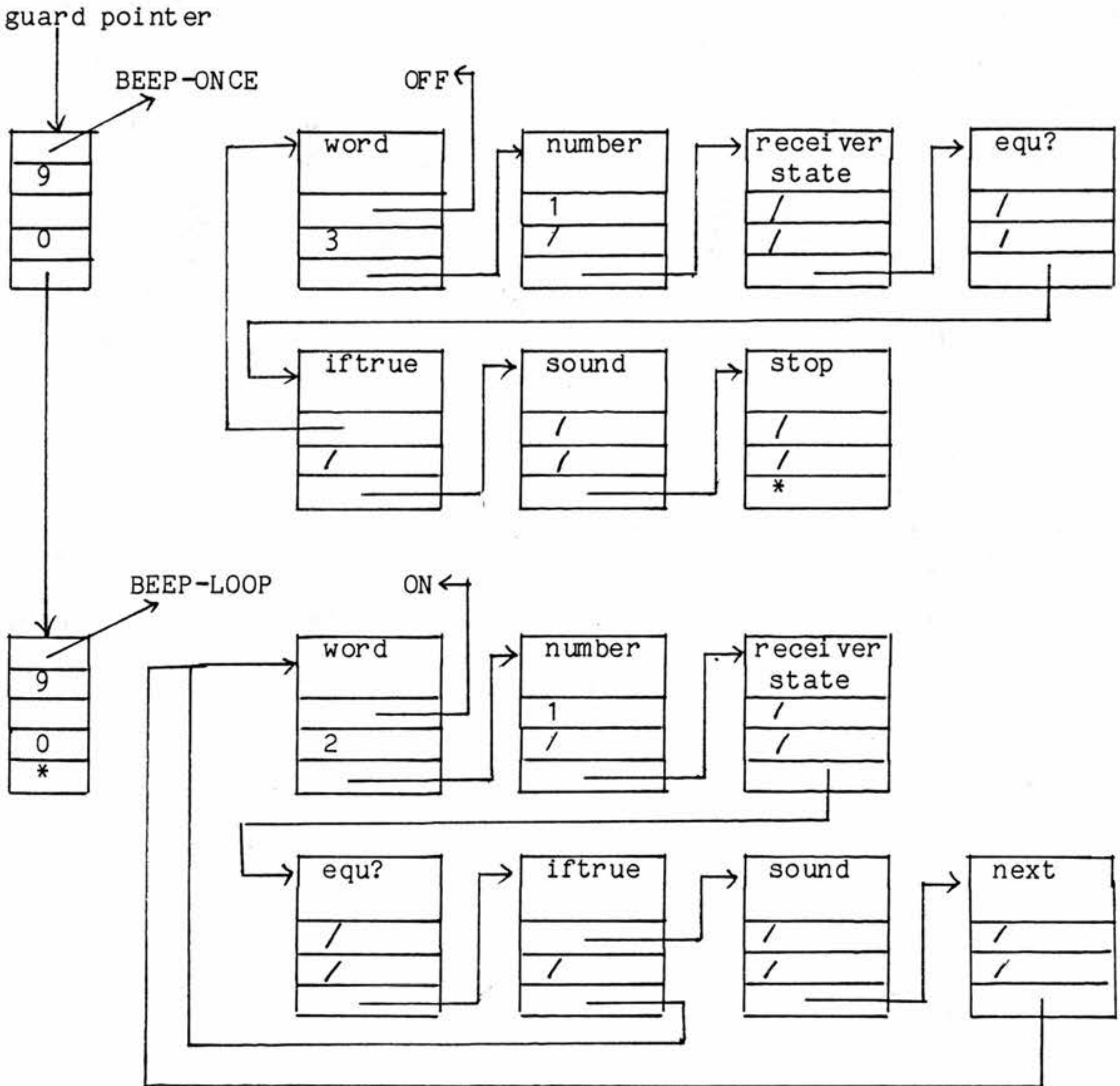
After the execution of the commands

```
TELL 'BEEP-ONCE
TELL 'BEEP-LOOP
```

and defined the guards DETECT-ONCE and DETECT-LOOP as

```
BEEP-ONCE;
WHILE EQU? 'OFF RECEIVER 1 ! STATE ();
SOUND

BEEP-LOOP;
WHENEVER EQU? 'ON RECEIVER 1 ! STATE (SOUND)
```

part of the internal memory looks like:



I.3 Run time organisation

The Concurrent-Logo system has eight local stacks. When a process is activated a free local stack is allocated to that process. This means that the system can handle up to eight processes in parallel. The following description first deals with the simple case of a single stack and a single process then followed by the description of the implementation of concurrency.

I.3.1 Local stack

A local stack is organised in records of three fields. A record may be used for

(1) storing intermediate results when evaluating expressions
(2) storing information concerning the caller of a procedure, which is used to collapse the stack by one level when the called procedure ends
(3) storing pointers to parameter/local variables

The latter two types of records are added to the stack whenever a user-defined procedure is called.

The different formats of a record are:

(1)

TYPE OF VALUE
VALUE
LENGTH OF VALUE

(2)

/
POINTER TO PROCEDURE DEFINITION OF CALLER
POINTER TO PREVIOUS STACK LEVEL

(3)

POINTER TO OBJECT'S NAME RECORD (if calling an object procedure
POINTER TO OBJECT'S OWN VARIABLES (if calling an object procedure)
POINTER TO LOCAL/PARAMETER VARIABLES

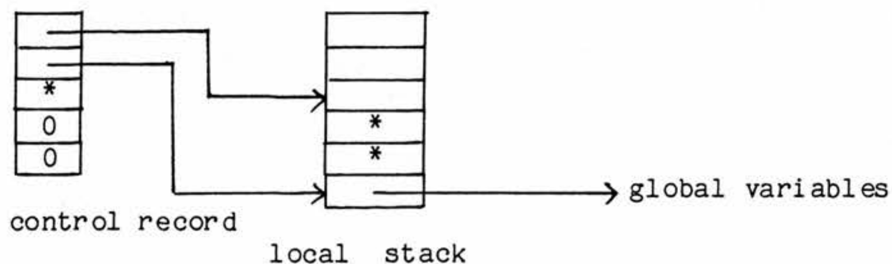
Associated with each stack is a record of control information. It has five fields, which provides the information concerning the current execution state of a process.

POINTER TO TOP OF STACK
POINTER TO BOTTOM OF STACK FOR CURRENT LEVEL
POINTER TO DEFINITION OF PROCEDURE
PROCESS NUMBER OF PARENT PROCESS
NUMBER OF CHILDREN PROCESSES

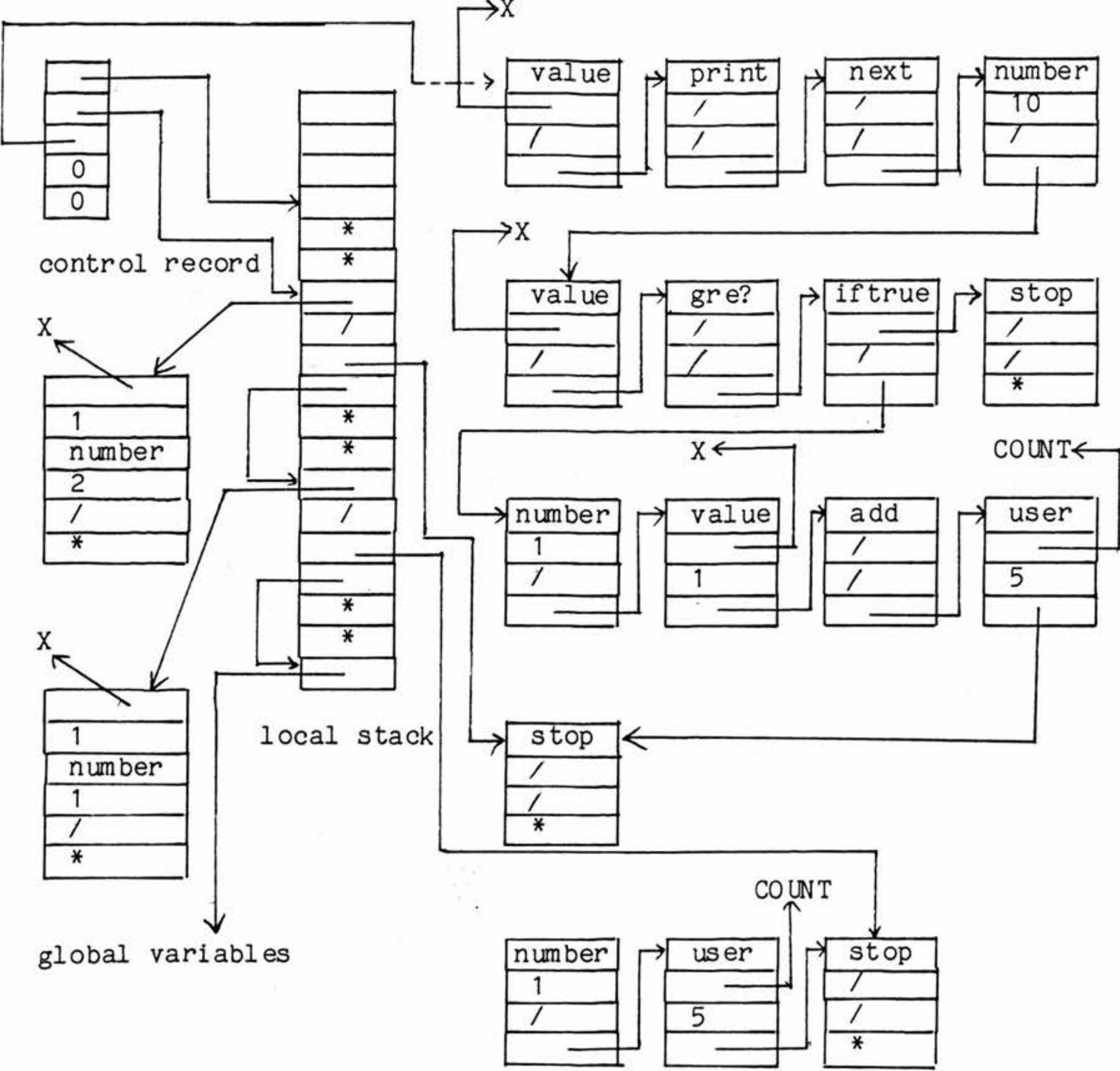
Consider the execution of the following procedure, which requires only one process:

```
COUNT 'X;
PRINT :X;
IF GRE? :X 10 (STOP)
   ELSE (COUNT ADD :X 1)
```

Initially the stack looks like:



After the procedure has called itself once (i.e recursed two levels) the stack looks like this:



I.3.2 Parallel processes

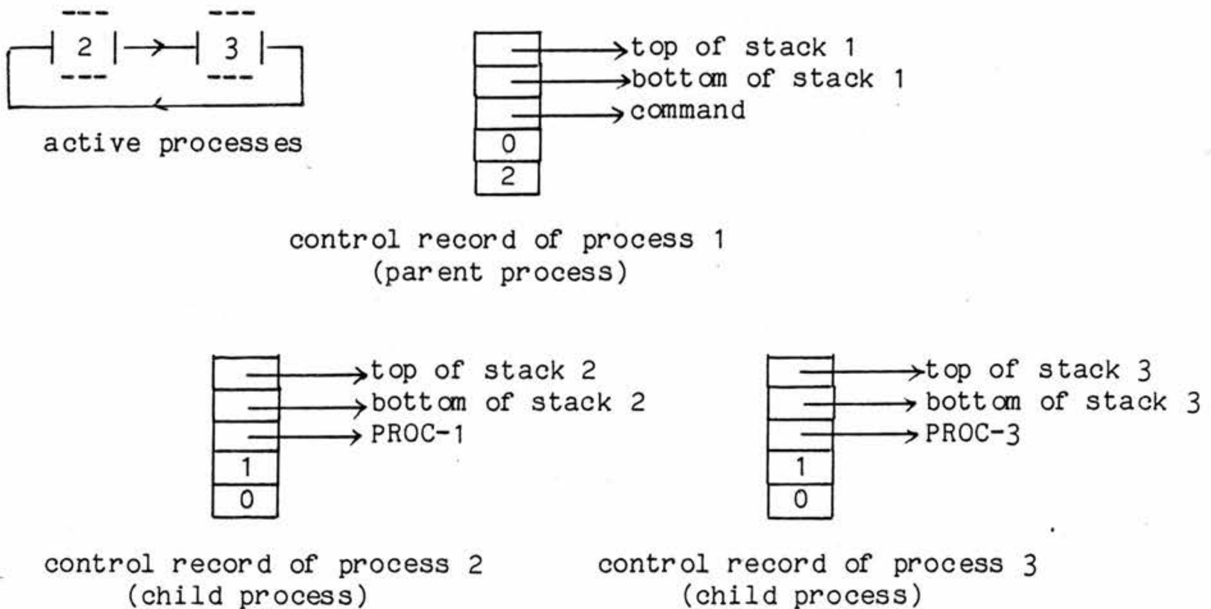
In Concurrent-Logo parallel processes can be activated by the parallel command '//'. A parallel command is completed when all the processes that it initiated have been completed.

Consider the execution of the command

```
PROC-1 // PROC-2 ; PRINT 'FINISHED
```

When executing the '//' command, the parent process creates two new processes (child processes): one for executing PROC-1 and the other for executing PROC-2. Once the child processes are created the parent process is suspended; i.e. only the child processes are active. When both child processes terminate, the execution of the parent process resumes and it prints the word FINISHED.

The implementation of this execution model is very simple. It requires the parent process to remember, in the control information record, the number of children processes it has created and each child process remembers the process number of the parent process. This can be shown diagramatically as follows:



When a child process terminates, its parent process's count of children is decremented by one. So when the count finally reached zero all of the children must have terminated and the parent process is made active again.

Concurrent-Logo provides only pesudo parallelism. When more than one process is active, the interpreter will switch process after every internal instruction is executed.

I.1.4 Syntax

The syntax of Concurrent-Logo is described using Backus-Naur form. Syntactic constructs are denoted by English word enclosed between the angular brackets < and > . These words also describe the nature and meaning of the constructs. The curly brackets { and } denotes repetition of the enclosed construct zero or more times. The symbol <empty> denotes the null sequence of symbols.

```
<commands>   ::= <command> {; <command>}
<command>    ::= <empty>                  | <procedure>              |
                 <if command>             | <whenever command>       |
                 <forever command>        | <guard command>          |
                 <object command>         | <repeat command>         |
                 <parallel command>       | <delayif command>        |
                 <while command>          | <define command>         |
                 <new class command>      | <new object command>
<procedure> ::= <procedure name> {<input>}
<if command>
            ::= IF <expression> (<commands>)                              |
                IF <expression> (<commands>) ELSE (<commands>)
<repeat command>
            ::= REPEAT <expression> (<commands>)
<while command>
            ::= WHILE <expression> (<commands>)
<forever command>
            ::= FOREVER (<commands>)
<whenever command>
            ::= WHENEVER <expression> (<commands>)
<parallel command>
            ::= <command> // <command>
<delayif command>
            ::= DELAYIF <expression>
<guard command>
            ::= <guard name> ! WAKEUP    | <guard name> ! SLEEP
<object command>
            ::= <system object command> | <user object command>
<user object command>
            ::= <object name> ! <procedure>
<system object command>
            ::= MOTOR <subscript> ! <motor command>
                RECEIVER <subscript> ! <receiver command>
                SWITCH <subscript> ! <switch command>
<motor command>
            ::= TURNC <input>           | TURNA <input>           |
                STOP                    | COUNT
<receiver command>
            ::= STATE                   | KEEPCOUNT               |
                STOPCOUNT               | COUNT                   |
                CLEARCOUNT
<switch command>
            ::= ON                      | OFF                     |
                ONUNTIL <input>         | STATE

<define command>
            ::= DEFINE <quoted word>    |
```

```
                 DEFINE <quoted word> CLASS <quoted word>
<new class command>
            ::= NEWCLASS <quoted word>  |
                NEWCLASS <quoted word> HAS {<quoted word>}
<new object command>
            ::= NEWOBJECT <quoted word> CLASS <quoted word>
<procedure definition>
            ::= <procedure name> {<argument>}; <commands>
<argument>  ::= <quoted word>
<input>     ::= <expression>
<expression>
            ::= <procedure>                  | <object_command>      |
                :<word>                      | <quoted word>         |
                <number>                     | <list>
<subscript> ::= <number>                     | :<word>
<procedure name>
            ::= <word>
<object name>
            ::= <word>
<variable name>
            ::= <word>
<guard name>
            ::= <word>
<integer>   ::= <digit>{<digit>}
<number>    ::= <integer>                    | <sign> <integer>
<sign>      ::= +                            | -
<word>      ::= <letter>{<character>}
<quoted word>
            ::= '<atom>
<atom>      ::= <character>{<character>}
<list>      ::= [ {<list element>} ]
<list element>
            ::= <number>                     | <list>                |
                <atom>
<digit>     ::= 0       | 1     | 2     | 3     | 4     | 5     |
                6       | 7     | 8     | 9
<character> ::= "       | #     | $     | %     | &     | ^     |
                +       | -     | =     | *     | ?     | _     |
                @
```

APPENDIX II

WORKSHEETS

WORKSHEET 1

THE BUFFER BOX

A description of the buffer box

On the top of the buffer box there are sixteen sockets; eight blue ones and eight yellow ones. The yellow ones are for sending out signals and therefore are called switches. The blue ones are for receiving signals and therefore are called receivers. In the near future you will learn how to operate these switches and receivers using a computer.

For the buffer box to work, it must be connected to the main power supply and a computer.

At one end of the buffer box there is a wire with a plug attached to its end. This is for plugging to the main power supply. There is also a switch besides it for switching the power on and off. At another side of the buffer box there are two sockets; one is marked 'INPUT' and the other 'OUTPUT'. These are for connecting the computer to the buffer box.

Connecting the computer to the buffer box

The type of computer that we are using is called a Terak. The ribbon cable that comes out from the back of the computer has two special plugs at its end. One is marked 'PORT 0' and the other 'PORT 1'. The plug that is marked PORT 0 should be connected to the socket that is marked INPUT on the buffer box. The plug that is marked PORT 1 should be connected to the socket that is marked OUTPUT.

Once all the connections are made and the power is switched on, you are then ready to start the computer running.

WORKSHEET 2

HOW TO USE THE BUFFER BOX

SWITCHes

The switches on the top of the buffer box are numbered 1 to 8. Just like most light switches, they can be switched ON and OFF. However, you cannot do that physically; you have to send them commands through the computer. The switches can be used to control lights or motors.

To turn a switch on: type

SWITCH n! ON <RETURN>

This sends a message telling switch n (a number between 1 and 8) to switch itself on. '<RETURN>' means press the key marked RETURN on the keyboard.

For example: to turn switch 1 on, type

SWITCH 1! ON <RETURN>

To turn a switch off: type

SWITCH n! OFF <RETURN>

For example: to turn switch 6 off, type

SWITCH 6! OFF <RETURN>

A switch can be in one of two states, ON or OFF. To find out the state of a switch: type

PRINT SWITCH n! STATE <RETURN>

This prints the state of switch n.

RECEIVERs

The receivers are for receiving signals from external sensors, e.g. button switches. A signal can be either ON or OFF. To find out the state of a receiver: type

PRINT RECEIVER n! STATE <RETURN>

Ideas to try out

(1) Using direct commands, make the computer switch a motor on and off.

(2) Make the computer detect the input from a button switch.

(3) Make the computer control the motor so that whenever the

button switch is pressed the motor will turn, and when the button switch is released the motor will stop turning.

WORKSHEET 3

CONTROLLING MOTORS

So far, you have used a computer to control a motor so that it can be made to turn or stop turning. A motor can do more than that, it can turn in different directions, clockwise or anticlockwise.

To control a motor's direction of rotation you need to connect the direction socket on the motor module to a switch on the buffer box. The motor is switched on and off as before, but the motor is made to turn in opposite directions by turning the direction switch on and off.

Ideas to try out

(1) Using direct commands, make a motor turn clockwise and anticlockwise.

(2) Make the computer detect signals from 3 button switches so that whenever

  (a) button switch 1 is pressed the motor turns clockwise

  (b) button switch 2 is pressed the motor turns anticlockwise

  (c) button switch 3 is pressed the motor stops.

WORKSHEET 4

THE IF COMMAND

You have already used the IF command in NOTE 1 and 2. Here is a more detailed description of how to use it.

The general form of the IF command is:

```
IF condition ( action ; action ; ......)
```

So the IF command is in three parts:

(1) the word IF, followed by

(2) a condition, followed by

(3) an action or a sequence of actions enclosed inside a pair of brackets.

The sequence of actions will be carried out only if the condition is true.

Conditions

A frequently used conditional test is to check whether two values are the same. You can do this by using the EQU? command.

EQU? takes two values as input. If the values are the same the word ´TRUE is returned, otherwise the word ´FALSE is returned.

For example, the command EQU? 2 2 returns the value ´TRUE. So if you type PRINT EQU? 2 2 to the computer it will print TRUE on the screen.

Try out the following on the computer:

```
PRINT EQU? 5 ADD 3 2

PRINT EQU? 6 MUL 3 2

PRINT EQU? 9 ADD 4 7

PRINT EQU? ´ON RECEIVER 1 ! STATE
```

Now hold down the button switch connected to RECEIVER 1 and try again the command

```
PRINT EQU? ´ON RECEIVER 1 ! STATE
```

Did the computer do what you expected it to do?

Now that you understand the EQU? command you can confidently use it

in the conditional part of the IF command.

Try out the following on the computer:

```
IF EQU? 2 2 ( PRINT [ MERRY CHRISTMAS ] )
```

In this command the condition is EQU? 2 2, which checks whether the numbers are equal, and the action is PRINT [MERRY CHRISTMAS]. Because the numbers are equal, therefore the command inside the brackets is obeyed, which prints the sentence MERRY CHRISTMAS.

```
IF EQU? 5 ADD 3 2 ( PRINT [ HAPPY NEW YEAR ] )
```

In this command the condition is EQU? 5 ADD 3 2, which checks whether 5 is equal to the result of adding 3 and 2. If they are the same the sentence HAPPY NEW YEAR is printed.

```
IF EQU? 10 ADD 4 5 ( PRINT [ THIS IS GREAT ] )
```

In this command the condition is EQU? 10 ADD 4 5, which checks whether 10 is equal to the result of adding 4 and 5. Because they are not equal, the command inside the brackets will not be carried out.

```
IF EQU? ´OFF RECEIVER 1 ! STATE (SOUND)
```

In this command the condition is EQU? ´OFF RECEIVER 1 ! STATE, which checks if the signal from RECEIVER 1 is ´OFF. If the signal is ´OFF then the action to be taken is to make a short beep.

```
IF EQU? ´ON RECEIVER 1 ! STATE (SWITCH 1 ! ON)
```

In this command the condition is EQU? ´ON RECEIVER 1 ! STATE, which checks if the signal from RECEIVER 1 is ´ON. If the signal is ´ON then the action to be taken is turn on SWITCH 1.

Work out the effect of the following command before tying it to the computer.

```
FOREVER ( IF EQU? ´ON RECEIVER 1 ! STATE (SOUND) )
```

Now type the command to the computer and see whether it does what you expected it to do.

WORKSHEET 5

PROCEDURES & THE WINDOW EDITOR

The most useful feature of Concurrent-Logo is the ease with which you can build new commands with procedures. Procedures can be used in exactly the same way as the commands built into Concurrent-Logo.

A procedure consists of two main parts:

TITLE LINE which contains the name of the procedure, followed by a semicolon, and

BODY which contains a series of commands. The commands are separated by semicolons and are executed in order.

For example:

```
WINDMILL;
FOREVER (
          IF EQU? ´ON RECEIVER 1! STATE
                      {MAKE THE MOTOR TURN CLOCKWISE}
                      (SWITCH 1! ON; SWITCH 2! ON);
          IF EQU? ´ON RECEIVER 2! STATE
                      {MAKE THE MOTOR TURN ANTICLOCKWISE}
                      (SWITCH 1! OFF; SWITCH 2! ON);
          IF EQU? ´ON RECEIVER 3! STATE
                      {STOP THE MOTOR}
                      (SWITCH 2! OFF)
        )
```

To make your procedures more readable
(1) choose meaningful procedure names,
(2) lay out commands neatly,
(3) include comments, if necessary.

A comment is English text surrounded by { and } symbols. A comment may appear anywhere in a procedure, may cover several lines and appear in the middle of a command, but may not contain the symbol }. All comments are ignored by the Concurrent-Logo and are used to amplify and explain the procedure to a human reader.

Examples of legal comments:

{ THIS IS A COMMENT } , { ANOTHER LEGAL COMMENT ^ % $ # [] }

Examples of illegal comments:

{ AN ILLEGAL COMMENT }} , { ANOTHER ILLEGAL COMMENT } OK}

These comments are illegal because, in both cases, the comments end after the first } symbol and all characters after it are treated as commands.

Primitives related to procedures

There are five primitives for dealing with procedures.

PROCEDURES
the names of all the procedures currently in the working memory are printed.

The following primitives take the name of a procedure as input.

DEFINE
it enters the window editor to allow you to define or change a procedure.
E.g. DEFINE WINDMILL

SCRAP
removes the named procedure from the working memory.
E.g. SCRAP WINDMILL

GET
fetches the named procedure from disk.
E.g. GET WINDMILL

LOSE
removes the named procedure from both disk and the working memory.
E.g. LOSE WINDMILL

Building and changing procedures

After you have typed the DEFINE command, the editor is invoked and you are in edit mode. The editor clears the text display, and puts a top and a bottom edge on it. A new cursor appears inside this window, and the number on the left side of the bottom edge indicates which line the cursor is sitting on. The keyboard controls the cursor until you finally presses the ETX or ESCAPE key to get back to the waiting prompt. The actions of defining or changing a procedure should be thought of as those of writing or amending a roll of paper (one roll per procedure) and the screen outlined should be thought of as a window onto the roll. The roll feeds in at the top, and out at the bottom. It can be moved both upwards and downwards so that text can disappear off the top or the bottom. You can change what is on the roll, or slip whole lines out of the roll, or add whole blank lines in. The maximum length of the roll is 44 lines long.

In edit mode, various keys that are normally not used now become useful. These allow you to amend text, insert or delete lines and move the roll up or down. The keys are:

The arrow keys
which move the cursor around the window in the direction suggested by the arrow.

BACKSPACE
deletes the character to the left of the cursor and causes the cursor to move to the left one space.

DEL
which deletes the character the cursor is standing on, and makes everything on the line to the right of the cursor move one space left.

US
which deletes the entire line on which the cursor is standing. Lower lines all move up.

LINEFEED
which pushes the line on which the cursor stands, and everything below, down a line i.e. it adds in a blank line. The cursor will then be sitting on the new blank line.

TAB
which is a toggle switch that enables or disables the INSERT MODE. In insert mode, when a character is typed, instead of overwriting the character that the cursor is sitting on the typed character is inserted; the cursor and all the characters which were underneath and to the left of it move one space to the right.

ETX
which causes the editor to store all of the procedure definition in working memory and on disk, and then return the system to waiting for the user's commands.

ESCAPE
which causes the editor to forget all the editing done in the current session and makes the system return to the waiting state.

Note that the RETURN key does not return the system to the waiting prompt, it only makes the cursor jump down to the start of the next line.

WORKSHEET 6

CONTROLLING A MECCANO TURTLE

The turtle is made out of Meccano with two DC motors mounted on it. The motor on the right hand side is used to drive the right wheel, and the motor on the left hand side is used to drive the left wheel.

You can make the turtle go forward, backward, left and right by controlling the motors. Use SWITCHes 1 and 2 to control the right motor, and use SWITCHes 3 and 4 to control the left motor. SWITCHes 1 and 3 are for controlling the motors' direction of rotation, and SWITCHes 2 and 4 are for turning the motors on and off.

Ideas to try out

(1) Drive the turtle using direct commands.

(2) Write four different procedures to make the turtle go forward, backward, left and right.

(3) Make the computer detect signals from five button switches so that whenever

(a) button switch 1 is pressed the turtle moves forward

(b) button switch 2 is pressed the turtle moves backward

(c) button switch 3 is pressed the turtle moves right

(d) button switch 4 is pressed the turtle moves left

(e) button switch 5 is pressed the turtle stops moving.

(4) Make the turtle move at different speeds.

(5) Make the turtle move in units of distance.

WORKSHEET 7

ARITHMETIC AND VARIABLES

Up to now, you have learnt to control SWITCHes and RECEIVERs, and have also learnt to use
(1) the FOREVER command,
(2) the IF command and
(3) the screen editor to build procedures.

In this note you will be introduced Concurrent-Logo´s arithmetic commands and the idea of a variable.

A first look at the PRINT command

Printing numbers:

e.g. PRINT 3

Try using the PRINT command with other numbers (both positive and negative).

Printing words:

e.g. PRINT ´HELLO

A word must have a single apostrophe before it.
Try using the PRINT command with other words.

Printing lists:

e.g. PRINT [THIS IS A LIST WITH 7 ELEMENTS]

Try using the PRINT command with other lists.

The PRINT command takes one input, which can be a number, word or list.

Arithmetic commands

Concurrent-Logo can do arithmetic for you. However, it only uses integers (whole numbers like -1000, 0, 2, 99).

For the operations of addition, subtraction, multiplication, division and finding the remainder, Concurrent-Logo uses the following commands:

	Concurrent-Logo commands
addition	ADD
subtraction	SUB

multiplication	MUL
division	DIV
remainder	REM

These arithmetic commands take two inputs and return a number as result.

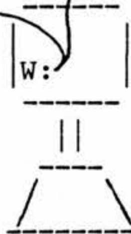
```
e.g. PRINT ADD 7 2
     PRINT SUB 7 2
     PRINT MUL 7 2
     PRINT DIV 7 2
     PRINT REM 7 2
```

Try out the above examples, and use Concurrent-Logo to evaluate the following arithmetic expressions:

a) 1 + 2 + 3 + 4
b) 1 - 2 + 3 - 4
c) 2 * 4 + 5
d) 4 / 2 + 5
e) 2 + 4 * 5
f) (2 + 9) * 3

As you have probably found out, the computer does not always understand what you type in. If it returned an unexpected result then try again.

I do not understand the expression 1 + 2 + 3,
but I understand ADD 1 ADD 2 3 ,
which has a result of 6.



Variables

Instead of printing the result of a computation on the screen, the value can be stored and looked at later. You do this by using the MAKE and VALUE commands.

```
E.g. MAKE 'EMERGENCY.NUMBER 999
     PRINT VALUE 'EMERGENCY.NUMBER
```

The MAKE command takes two inputs. The first is the name (a word) of a box, and the second is the thing that is to be put into the box.

The VALUE command takes one input, which is the name (a word) of a box, and returns the thing that is inside the box.

Note that colon (:) may be used as an abbreviation for the VALUE command. If used, the colon must be the character immediately before the name of the box, and the single apostrophe is omitted.
E.g. PRINT :EMERGENCY.NUMBER

Try out the following on a computer:

```
MAKE ´FRUIT ´APPLE
MAKE ´APPLE ´DELICIOUS
PRINT :FRUIT
PRINT :APPLE

MAKE ´APPLE.PRICE 30
MAKE ´PORK.PRICE 95
MAKE ´BREAD.PRICE 40
PRINT :APPLE.PRICE
PRINT :PORK.PRICE
PRINT :BREAD.PRICE
MAKE ´TOTAL.PRICE ADD :APPLE.PRICE ADD :PORK.PRICE :BREAD.PRICE
PRINT :TOTAL.PRICE
```

WORKSHEET 8

MANIPULATING LISTS

In the previous lesson you have learnt how to print a list. In fact, there is a lot more you can do with lists. This note tells you something about manipulating lists.

What a list is

A list is an ordered collection of numbers and words. The collection is bound between two square brackets, [at the start,] at the end. These are lists:

```
[ THIS IS A LIST ]
[ 0 1 2 3 4 5 6 7 8 9 ]
[ THERE ARE 10 GREEN BOTTLES ON THE WALL ]
[]
```

In the first, second and third examples, the words and the numbers are called the ´elements´ of the list.

The fourth example was the **empty list.** It has no (0) elements.

Inside a list, words do not need to begin with a quote mark.

LENGTH

To find out how many elements there are in a list, use the LENGTH primitive, like this:

```
PRINT LENGTH [ HA HA HA ]
PRINT LENGTH [ HELLO HELLO HELLO HELLO HELLO ]
PRINT LENGTH [ HOW ARE YOU ? ]
PRINT LENGTH [ HOW ARE YOU? ]
PRINT LENGTH []
```

These commands will make Concurrent-Logo print 3, 5, 4, 3 and 0 respectively Try them on the computer.

FIRST and REST

The primitive FIRST finds the first item in a list and returns it. For example if you gave the commands

```
MAKE ´X [ SUNDAY WAS VERY QUIET ]
```

and

```
MAKE ´Y [ MORNING HAS BROKEN ]
```

then

```
PRINT FIRST :X
PRINT FIRST :Y
```

will print respectively

```
SUNDAY
MORNING
```

Whatever is first in the list is passed back to PRINT.

The primitive REST removes the first item from a list and returns the rest of the list. For example

```
PRINT REST :X
PRINT REST :Y
```

will print

```
WAS VERY QUIET
HAS BROKEN
```

You may not use the empty list as input to FIRST and REST (if you do, you get an error message).

Adding elements to a list

You can tack an element onto the begging of a list using PUTF (put first). Here is an example of PUTF:

```
MAKE ´X [ 2 3 4 ]
PRINT PUTF 1 :X
```

Concurrent-Logo prints

```
1 2 3 4
```

but (as before) X is still [2 3 4].

To make X become [1 2 3 4]:

```
MAKE ´X PUTF 1 :X
PRINT :X
```

Example and exercise

Example: Print a short description of a friend.

```
FRIEND ´NAME ´PRONOUN;
PRINT PUTF :NAME [ IS A FRIEND OF MINE ];
PRINT PUTF :PRONOUN [ IS NOW 16 ];
PRINT PUTF :PRONOUN [ IS VERY TALL ];
```

This procedure takes two inputs. The first is a name of a friend. The second is a word, either HE, SHE or IT.

Try it out with

```
FRIEND 'JIMMY 'HE
FRIEND 'SUSAN 'SHE
FRIEND 'BOBBY 'IT
```

Exercise:

A procedure called PUTL (put last) is defined for you, and is on your disk. This procedure is similar to PUTF, but instead of tacking an element onto the beginning of a list, it tacks the element onto the end of a list.

For example

```
PRINT PUTL 'THIN PUTL [ HE IS VERY ]
```

would print HE IS VERY THIN.

Get the procedure PUTL into memory and then change the FRIEND procedure so that it can describe your friends differently.

WORKSHEET 9

CONTROLLING A MECCANO LIFT

The lift is made out of Meccano with one DC motor and three reed switches. The motor is used to drive the lift cage up and down, and the reed switches are for detecting whether the lift cage is at a particular location.

Use switch 1 for turning the motor on and off, and switch 2 for controlling the motor's direction of rotation. When the motor turns clockwise it winds the string that is attached to the lift cage so that the lift cage is pulled upwards; and when the motor turns anticlockwise it unwinds the string so that the lift cage will move downwards.

Connect the bottom reed switch to RECEIVER 1, and the second and top reed switches to RECEIVERs 2 and 3 respectively. Initially the states of all these RECEIVERs are OFF. When the lift cage passes any of these reed switches the corresponding RECEIVER's state will then be ON.

By controlling the motor, you can make the lift cage go to different floors. You can know whether the lift cage has reached a particular floor by detecting the state of the corresponding RECEIVER.

Programs for you to try out

Type the following to the computer

```
GETCLASS LIFTS
NEWOBJECT LIFT CLASS LIFTS
LIFT! READY
```

The above three commands set up the lift for you.

You can make the lift cage move up to the third floor and down to the ground floor by the command

```
LIFT! UPANDDOWN
```

Now try

```
REPEAT 2 ( LIFT! UPANDDOWN )
```

You can make the lift cage move to a particular floor by giving the lift the MOVETO command. The MOVETO command takes one input, which is the floor number, e.g. 1 for ground floor. Try the following:

```
LIFT! MOVETO 3
LIFT! MOVETO 2
LIFT! MOVETO 1
LIFT! MOVETO 3; LIFT! MOVETO 1
```

Exercise

(1) Write you own procedure CYCLE which moves the lift cage up and down repeatedly.

(2) Write you own procedure GOTO which moves the lift cage to a particular floor.

(3) Make the computer detect signals from three button switches so that whenever

   (a) button switch 1 is pressed the lift cage moves to the ground floor.

   (b) button switch 2 is pressed the lift cage moves to the second floor.

   (c) button switch 3 is pressed the lift cage moves to the third floor.

WORKSHEET 10

A COMPUTER CONTROLLED SECURITY SYSTEM

The idea of this project is to learn something about how a computer can be used to protect a house against burglars.

Connections

The Meccano doll's house has four micro-switches, two reed switches, one button switch and one DC motor fitted to it.

Use switch 1 for turning the motor on and off, and switch 2 for controlling the motor's direction of rotation. When the motor turns anticlockwise the door slides open; when the motor turns clockwise the door slides to its close position.

Looking at the house from the back, connect

(1) the right reed switch to RECEIVER 1 and

(2) the left reed switch to RECEIVER 2.

The reed switches are used to detect the position of the door. When the door is at the closed position the state of RECEIVER 1 is OFF and RECEIVER 2 is ON; when the door is at the open position the state of RECEIVER 1 is ON and RECEIVER 2 is OFF.

Behind every window on the doll's house there is a micro-switch. Again, looking at the house from the back, connect

(1) the top left micro-switch to RECEIVER 3,

(2) the bottom left micro-switch to RECEIVER 4,

(3) the top right micro-switch to RECEIVER 5 and

(4) the bottom right micro-switch to RECEIVER 6.

Initially the states of these RECEIVERs are OFF. When a window is pushed opened the corresponding RECEIVER's state will then be ON.

Connect the button switch, which is next to the sliding door, to RECEIVER 7. The button switch is used as a door bell.

Ideas to try out

(1) Make the computer detect if a burglar is trying to break in through any of the windows.

(a) If any of the windows is open, make the computer sound continuously and print the message [SOMEONE IS TRYING TO BREAK IN THROUGH THE WINDOWS].

(b) Modify the program so that the computer tells you exactly which window is being opened.

(2) Make the computer be the door keeper.

(a) Whenever the button switch is pressed the door slides open, and then after a short while the door closes automatically.

(b) Whenever the button switch is pressed the computer asks you for a secret word. If your answer is correct the door slides open, otherwise the door remains closed.

(3) Combine the ideas above in one program.

WORKSHEET 11

TURTLE AND STEPPING MOTORS

The turtle you are going to work with is a modified version of the previous one. The main difference is the type of motors used.

Connect the left motor to the socket marked ´1´ on the stepping motor module; connect the right motor to the socket marked ´2´. Now, the left motor is referred to as MOTOR 1; the right motor is referred to as MOTOR 2. Each motor can respond to 3 commands:

(1) TURNC n, which tells a motor to turn clockwise n steps.

For example, the command

```
MOTOR 1! TURNC 200
```

would make the left motor turn clockwise 200 steps.

(2) TURNA n, which tells a motor to turn anti-clockwise n steps. For example, the command

```
MOTOR 2! TURNA 200
```

would make the right motor turn anti-clockwise 200 steps.

(3) STOP, which tells a motor to stop turning. For example, the command

```
MOTOR 2! STOP
```

would make the right motor stop turning.

Making the turtle move forward

Here is a procedure that would make the turtle move forward.

```
FORWARD ´X;
MOTOR 1! TURNC :X // MOTOR 2! TURNA :X
```

The procedure takes a number as input. The input value specifies the number of steps that the turtle is to be moved. When the procedure is called, the two motor commands will be run in parallel, causing the motors to turn simultaneously. Notice, the motors are mounted back to back. Therefore, for the turtle to move in a straight line the two motors have to turn in opposite directions.

Exercise

(1) Type in the FORWARD procedure and try it out with different input values.

(2) What would happen if the symbol ´//´ is replaced by ´;´?

(3) Define your own procedures: BACK, LEFT and RIGHT.

WORKSHEET 12

TURTLE AND REFLECTIVE OPTO-SWITCHES

There are two small black objects fixed to the front of the turtle. They are called reflective opto-switches (ROS). Connect the left ROS to RECEIVER 1 and the right ROS to RECEIVER 2. Once a ROS is connected to the BUFFER BOX,

(1) when it is placed above a black surface, it sends an ´ON´ signal to the computer;

(2) when it is placed above a reflective surface, for example a white surface, it sends an ´OFF´ signal to the computer.

The procedure

```
LEFT.ON.WHITE;
RESULT NOT EQU? ´ON RECEIVER 1! STATE
```

would return the value ´TRUE if the left ROS is on a reflective surface, otherwise ´FALSE.

Exercise

(1) Type in the LEFT.ON.WHITE procedure and try it out.

(2) Define your own procedure RIGHT.ON.WHITE to test whether the right ROS is on a white surface.

(3) You now have a collection of procedures that would

(a) make the turtle move in different directions for a specified number of steps;

(b) test whether the turtle is on a white surface.

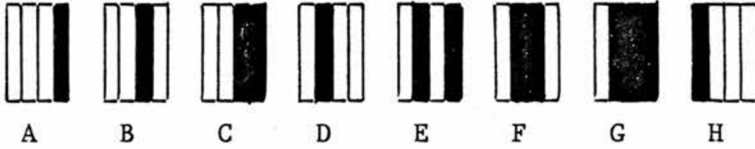
Making use of these procedures, write a new procedure that would make the turtle follow a black line on a white surface.

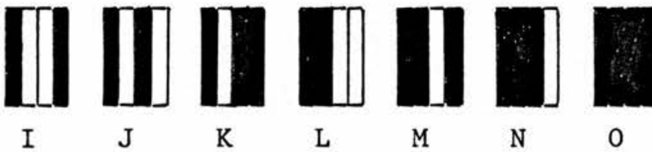
Hint: make the turtle move forward in small steps,

WORKSHEET 13

BINARY CODE

Let the letters A to O be represented by the following patterns:





Each pattern has four binarys. Each binary is either black or white. We shall call these patterns **binary codes**. You may have seen similar binary codes on products you buy from super-markets. Using the given information above, can you work out what the following sequence of binary codes represents:

Answer:_________________________.

I am sure you found the problem very easy. Let us consider how we can program the computer to do it for us.

Identifying an 'A'

We are going to make use of the turtle with reflective-opto switches.

The binary code for ´A´ is drawn on a piece of paper. The width of each binary is 21 units of turtle movement, i.e. FORWARD 21 would move the turtle from the centre of one binary to the centre of the next binary. The following procedure A would make the turtle move over a binary code and print out whether it represents ´A´.

```
A;
MAKE ´ANSWER ´TRUE;
FORWARD 21;
IF LEFT.ON (MAKE ´ANSWER ´FALSE);
FORWARD 21;
IF LEFT.ON (MAKE ´ANSWER ´FALSE);
FORWARD 21;
IF LEFT.ON (MAKE ´ANSWER ´FALSE);
FORWARD 21;
IF LEFT.OFF (MAKE ´ANSWER ´FALSE);
IF :ANSWER (PRINT [THE BAR CODE IS ´A´])
      ELSE (PRINT [THE BAR CODE IS NOT ´A´])
```

The procedure assumes that the binary code does represent the letter ´A´, That is why the first command sets the variable ANSWER to TRUE. It then makes the turtle travel from one binary to the next. If it detects that the colour of a binary is not as expected it changes the value of ANSWER to FALSE. After all the binarys have been tested if the value of ANSWER remains TRUE then the binary code does represent A.

Exercise

(1) Type in the procedure A and try it out.

(2) Define procedures to identify other binary codes.

(3) Define a procedure to tell you what a binary code represents.

WORKSHEET 14

A TEACHING PROGRAM FOR THE ROBOT ARM

The robot arm you are going to work with is called ARMDROID. It consists of five main parts: fingers, wrist, forearm, upper arm and shoulder. The following shows the movements that the different parts can make:

(1) fingers can open and close;

(2) wrist can raise up and lower down vertically;

(3) wrist can rotate right or left;

(4) forearm can rotate clockwise and anti-clockwise, about a horizontal axis on the upper arm;

(5) upper arm can rotate clockwise and anti-clockwise, about a horizontal axis on the shoulder;

(6) shoulder can rotate clockwise and anti-clockwise, about the base.

Notice the wrist has two sets of movements.

There is a set of procedures, already defined for you, for manipulating ARMDROID. They are called TEACH, REMEMBER, REPLAY and FORGET.

The procedure TEACH

This procedure enables you to control the movement of ARMDROID by single key-presses. As described above, there are six sets of movements, with two directions in each set, associated with the ARMDROID. Therefore, twelve keys on the keyboard are used to initiate movement of the arm. Fig 1 represents part of the TERAK keyboard, and the keys that are used are bracketed.

```
 ---------------------------------------
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0 |
 ---------------------------------------
  |(Q)|(W)|(E)|(R)|(T)|(Y)| U | I | O | P |
  ----------------------------------------
   |(A)|(S)|(D)|(F)|(G)|(H)| J | K | L |
   ------------------------------------
```

fig 1

The function of the keys

(1) The two keys ´Q´ and ´A´, below the numeric key 1, are for opening and closing the fingers respectively.

(2) The two keys ´W´ and ´S´, below the numeric key 2, are for raising and lowering the wrist respectively.

(3) The two keys ´E´ and ´D´, below the numeric key 3, are for rotating the wrist.

(4) The two keys ´R´ and ´F´, below the numeric key 4, are for rotating the forearm.

(5) The two keys ´T´ and ´G´, below the numeric key 5, are for rotating the upperarm.

(6) The two keys ´Y´ and ´H´, below the numeric key 6, are for rotating the base.

All other keys, when pressed, stop the arm moving.

When the procedure is first called, it asks the question

FOR EACH COMMAND, HOW MANY UNITS OF MOVEMENT ?

Type in a number and press RETURN. From then on, whenever you press any of the tweleve keys mentioned above, the corresponding part of the arm will move, in the predefined direction, the number of steps input. To change the number, press RETURN and you will be asked the question again.

The idea is to move the arm, in big steps, roughly to its desired position. Then, move it in small steps to the exact position.

Once you have moved the arm to the desired position, press ESC to stop the procedure.

The procedure REMEMBER

The procedure REMEMBER tells the computer to remember the present position of ARMDROID.

The procedure is defined to remember up to ten positions in the order you told the computer.

The procedure REPLAY

The procedure REPLAY does two things:
(1) it moves ARMDROID to the starting position;
(2) it moves ARMDROID from position to position in the order you told the computer to remember.

The procedure FORGET

The procedure FORGET tells the computer to forget the sequence of positions it has been told to remember previously.

The set of procedures just described allows you to train ARMDROID to follow a sequence of actions.

Ideas to try out

(1) Train ARMDROID to pick up a block and put it on top of another.

(2) Train ARMDROID to put a few blocks into a box and then put the lid on the box.

WORKSHEET 15

HOW ARMDROID WORKS

The five main parts of ARMDROID are controlled by six stepping motors:

(a) MOTOR 1 controls the fingers
(b) MOTOR 2 and 3 control the wrist
(c) MOTOR 4 controls the forearm
(d) MOTOR 5 controls the upper arm
(e) MOTOR 6 controls the shoulder.

The stepping motors used for ARMDROID and the turtle are of the same kind.

To recap, the commands for stepping motors are

(a) TURNC n, which tells a motor to turn clockwise n steps.
E.g. MOTOR 1! TURNC 200.

(b) TURNA n, which tells a motor to turn anti-clockwise n step.
E.g. MOTOR 2! TURNA 200.

(c) STOP, which tells a motor to stop turning.

Now, send commands to different motors and observe the effect on ARMDROID.

Closing the fingers

Here is a procedure that would close the fingers:

```
FINGERS.CLOSE ´X;
MOTOR 1! TURNC :X
```

The procedure takes a number as input. The input value specifies the number of units that the fingers are to be closed.

Exercise

(1) Type in the FINGERS.CLOSE procedure and try it out with different input values.

(2) Define procedures FINGERS.OPEN, FOREARM.DOWN, FOREARM.UP, UPPERARM.DOWN, UPPERARM.UP, SHOULDER.LEFT, and SHOULDER.RIGHT.

Controlling the wrist

Unlike other parts, the wrist is controlled by two motors. By trying out the following commands and filling in the blanks you will appreciate how the wrist is being controlled.

(1) The command MOTOR 2! TURNA 50
made the wrist move ____________________________.

(2) The command MOTOR 2! TURNC 50
made the wrist move ____________________________.

(3) The command MOTOR 3! TURNA 50
made the wrist move ____________________________.

(4) The command MOTOR 3! TURNC 50
made the wrist move ____________________________.

(5) The command MOTOR 2! TURNA 50 // MOTOR 3! TURNA 50
made the wrist move ____________________________.

(6) The command MOTOR 2! TURNA 50 // MOTOR 3! TURNC 50
made the wrist move ____________________________.

Exercise

Define procedures WRIST.UP, WRIST.DOWN, WRIST.RIGHT and WRIST.LEFT.

WORKSHEET 16

MORE ABOUT HOW ARMDROID WORKS

From the previous note you have learnt how different parts of ARMDROID are controlled by stepping motors. This note tells you how the computer keep track of ARMDROID's position.

Keeping count

Every stepping motor has a variable COUNT. When the computer is switched on these variables are set to 0.

Whenever a motor turns clockwise the value of its COUNT variable is automatically incremented by the number of steps turned; whenever a motor turns anticlockwise the value of its COUNT variable is decremented by the number of steps turned. For example, if the value of MOTOR 1's COUNT is 0, after executing the command

```
MOTOR 1! TURNC 50
```

the value of MOTOR 1's COUNT would be 50. The values of other MOTORs' COUNT variables are unaffected. If we now give the command

```
MOTOR 1! TURNA 80
```

the value of MOTOR 1's COUNT would be decremented by 80, and it would have -30 as its new value.

The command

```
MOTOR n! COUNT
```

returns the value of MOTOR n's COUNT.

E.g. PRINT MOTOR 1! COUNT

would print -30 on the screen.

ARMDROID's position

Any position within the space of ARMDROID's movement can be conveniently representing as a list of six numbers. The starting position of ARMDROID is always [0 0 0 0 0 0], the first number being the value of MOTOR 1's COUNT, the second being the value of MOTOR 2's COUNT and so on. For example, to move ARMDROID from its starting position to position [20 0 0 0 -100 0], MOTOR 1 needs to turn clockwise 20 steps and MOTOR 5 needs to turn anticlockwise 100 steps.

There are two useful procedures which are already defined for you. They are called: POSITION and MOVETO.

The procedure POSITION

This procedure returns the position of ARMDROID as a list of six

numbers. For example, if ARMDROID was at its starting position, the command

```
PRINT POSITION
```

would print 0 0 0 0 0 0 on the screen. If MOTOR 1 had turned anticlockwise 30 steps and MOTOR 4 had turned clockwise 100 steps then the command

```
PRINT POSITION
```

would print -30 0 0 100 0 0 on the screen.

The procedure MOVETO

The procedure MOVETO takes a list of six numbers as input. When the procedure is called, it moves ARMDROID to the position specified by the input.

For example, if you had recorded a block is at position [-20 30 245 -68 97 100] then the command

```
MOVETO [-20 30 245 -68 97 100]
```

would move ARMDROID to the block. The command

```
MOVETO [0 0 0 0 0 0]
```

would move ARMDROID to its starting position.

Exercise

(1) Define a procedure PICKUP that would move ARMDROID to pick up a block at a particular position.

(2) Define a procedure PUT.IN.BOX that would put whatever is in ARMDROID's fingers into a box.

APPENDIX III

QUESTIONNAIRES

QUESTIONNAIRE 1 NAME

Please fill in the following:

(1) Have you used a computer before?

If yes, what did you use it for?

(2) Do you own or use a computer now?

(3) Write down all the things that you know or think a computer can do?

(4) Have you written any program before?

If yes, please write down brief descriptions of the programs and the programming languages used.

(5) Can you program the computer to do any of the things you mentioned in 3)?

(6) Do you think computing is fun? If you have not done any, does it sound like fun?

(7) Do you think computing is important? Try to say why, not just ´yes´ or ´no´.

(8) Would you like to know more about computing?

(9) Do you think you would be good at computing?

QUESTIONNAIRE 2 NAME:

A 'robot' is really any kind of controllable machine that you don't have to work directly. Besides the science-fiction notion of robot, there are examples such as various radio-controlled toys, 'Big Trak' and 'programmable' train sets.

Please fill in the following:

(1) Do you think robots are fun?

(2) Do you have some idea of how a robot works?

(3) Would you like to learn to program a robot?

(4) Do you like building working models of any kind?
If you do, please write down what you use for building - e.g. Meccano, Fisher-Technik, wood.

QUESTIONNAIRE 3 NAME:

INTERMEDIATE PROGRESS SURVEY

I would like to know something about your response towards the course so far. I have set out some questions to show you the kind of information that I need. I would be very interested in any additional comments and suggestions that you make.

General questions

For the following 3 questions, please put a circle around the most appropriate answer.

(1) Do you think you have learned anything useful so far?
 (a) a lot
 (b) quite a lot
 (c) some
 (d) a little
 (e) very little

(2) Did you find the course enjoyable?
 (a) most enjoyable
 (b) enjoyable
 (c) fair
 (d) boring
 (e) very boring

(3) Do you think your teacher had given you the help you required?
 (a) far too much
 (b) too much
 (c) about right
 (d) too little
 (e) much too litle

Concerning each device

So far you have worked with the following devices:
 (1) Windmill
 (2) Turtle
 (3) Doll's house
 (4) Lift
Which one did you enjoy working with most? Why?

For each device please describe (you may consult your notes):
 (1) What are the essential components? What are they for?
 (2) What did you learn through working with it?
 (3) What did you find difficult?
 (4) What did you find easy?
 (5) What did you find interesting?
 (6) Would you like to have spent more, or less, time with it?

QUESTIONNAIRE 4 NAME:

FINAL SURVEY

I would like to know something about your response towards the whole course. I have set out some questions to show you the kind of information that I need. I would like to stress that this is not a test so please feel free to express your own opinion. I would be very interested in any additional comments and suggestions that you make.

For a multiple choice question please put a circle around the most appropriate answer. For other questions please put your answers on the blank papers provided.

(1) Did you find the course enjoyable?
- (a) most enjoyable
- (b) enjoyable
- (c) fair
- (d) boring
- (e) very boring

(2) Do you think you have learned anything useful?
- (a) a lot
- (b) quite a lot
- (c) some
- (d) a little
- (e) very little

(3) Do you think your teacher (Paul Chung) had given you the help you required?
- (a) far too much
- (b) too much
- (c) about right
- (d) too little
- (e) much too litle

(4) Did you find the notes helpful?
- (a) very helpful
- (b) helpful
- (c) fair
- (d) not helpful
- (e) not helpful at all

(5) Did you find the notes clear?
- (a) very clear
- (b) clear
- (c) fair
- (d) not clear
- (e) not clear at all

(6) How many people do you prefer to work with?
(a) by yourself
(b) with 1 friend
(c) with 2 friends
(d) with 3 friends
(e) with 4 friends

(7) Would you like to have received more notes?

(8) Would you like your teacher (Paul Chung) to give you more explanation and teaching about computer control applications?

(9) Would you recommend the course to your friends?

(10) So far you have worked with the following devices:
(a) Windmill
(b) Turtle
(c) Doll's house
(d) Lift
(e) Turtle with reflective-opto sensors
(f) Robot arm

Please write them down in order of preference, starting with the one you enjoy working with most.

(11) For the top three choices, give reasons for why you like them.

(12) For the version of the turtle with sensors and the robot arm please describe (you may consult your notes):

(a) What are the essential components? What are they for?
(b) What did you learn through working with it?
(c) What did you find difficult?
(d) What did you find easy?
(e) What did you find interesting?
(f) Would you like to have spent more, or less, time with it?

(13) Would you like to design your own device, e.g. crane, using Meccano and then write a program in Concurrent-Logo to control it?

(14) Do you feel confident that you can do it? What are the difficulties?

(15) What other devices would you like to work with?

(16) Would you like to have Concurrent-Logo on your personal computer? Why?

(17) Did you find Concurrent-Logo easy to use?

(18) What did you like about it?

(19) What didn't you like about it?

(20) Would you like to learn more about Concurrent-Logo?

(21) To make stepping motor 1 turn clockwise 300 steps, which command do you prefer (or suggest your own)?

(a) MOTOR 1 ! TURNC 300
(b) MOTOR 1 TURNC 300
(c) TURNC 1 300

(22) To turn switch 2 on, which command format do you prefer (or suggest your own)?

(a) SWITCH 2 ! ON
(b) SWITCH 2 ON
(c) ON 2

(23) To find out the state of receiver 3, which command format do you prefer (or suggest your own)?

(a) RECEIVER 3 ! STATE
(b) RECEIVER 3 STATE
(c) STATE 3

QUESTIONNAIRE 5 NAME:

TEST

I would like to know how much you have learnt from the course. I have set out some questions for you to answer. Please try to answer all questions and to give the fullest explanation.

(1) What is a push-button switch for?

(2) What is a reed switch for?

(3) What is a reflective-opto switch for?

(4) What are the differences between working with DC motors and stepping motors?

(5) What are the difficulties in training a robot arm to do a sequence of actions?

(6) What are the advantages of having sensors attached to a control device?

(7) Please describe the effects of

```
(a) PRINT ´A
(b) REPEAT 10 (PRINT ´A)
(c) FOREVER (PRINT ´A)
(d) REPEAT 10 (PRINT ´A); REPEAT 10 (PRINT ´B)
(e) REPEAT 10 (PRINT ´A) // REPEAT 10 (PRINT ´B)
(f) FOREVER (PRINT ´A); FOREVER (PRINT ´B)
(g) FOREVER (PRINT ´A) // FOREVER (PRINT ´B)
```

(8) What is the difference between?

(a)

```
IF EQU? :A :B (SOUND)
```

and

```
FOREVER (IF EQU? :A :B (SOUND))
```

(b)

```
FOREVER (PRINT ´A; PRINT ´B; PRINT ´C; SOUND)
```

and

```
FOREVER (PRINT ´A; PRINT ´B; PRINT ´C) // FOREVER (SOUND)
```

(c)

```
FOREVER (IF EQU? 'ON RECEIVER 1! STATE (REPEAT 1000 (PRINT 1);
         IF EQU? 'ON RECEIVER 2! STATE (REPEAT 1000 (PRINT 2);
         IF EQU? 'ON RECEIVER 3! STATE (REPEAT 1000 (PRINT 3))
      and

FOREVER (IF EQU? 'ON RECEIVER 1! STATE (REPEAT 1000 (PRINT 1)) //
FOREVER (IF EQU? 'ON RECEIVER 2! STATE (REPEAT 1000 (PRINT 2)) //
FOREVER (IF EQU? 'ON RECEIVER 3! STATE (REPEAT 1000 (PRINT 3))
```

(9) Do you think procedures are useful? If yes, why?

(10) If you know BASIC, can you tell me what are the differences between subroutines in BASIC and procedures in Concurrent-Logo?

(11) Do you consider parallelism is an important part of a programming language? Why?

(12) Fig 1 is a model cable car made out of Meccano. I would like you to consider using a computer to control it. Assuming the cable car is initially at the bottom level, the task is

(1) make the cable car move to the top level

(2) wait for a short while

(3) make the cable car move to the bottom level

(4) wait for a short while

(5) continue from 1.

What electronic components would you use for this project?

What are you using them for?

Please write down the names of the procedures you would define, and explain the function of each (no need to write the procedure code).

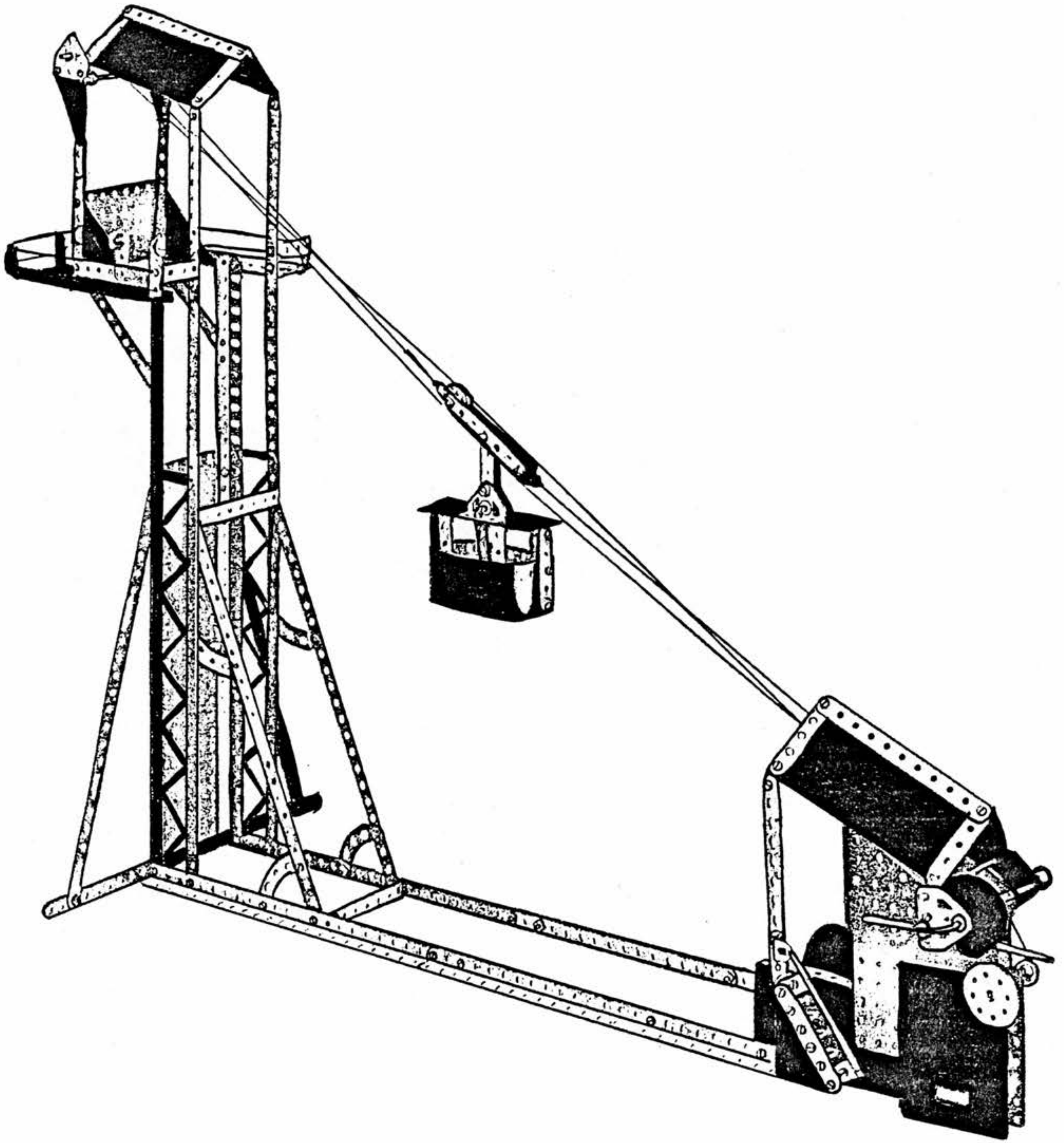


Figure 1 Cable car

(13) Fig 2 is a model crane made out of Meccano. It has three moving parts:

(a) the hook which moves up and down

(b) the jib which rotates clockwise and anticlockwise in a vertical plane

(c) the upper part of the crane which rotates clockwise and anticlockwise in a horizontal plane.

I would like you to consider using a computer to control it. The task is to provide a control box with six buttons so that whenever

(1) button 1 is pressed the hook moves up a fixed amount

(2) button 2 is pressed the hook moves down a fixed amount

(3) button 3 is pressed the jib rotates clockwise a fixed amount

(4) button 4 is pressed the jib rotates anticlockwise a fixed amount

(5) button 5 is pressed the upper part of the crane rotates clockwise a fixed amount

(6) button 6 is pressed the upper part of the crane rotates anticlockwise.

What electronic components would you use?

What are you using them for?

Please write down the names of the procedures you would define, and explain the fuction of each (no need to write the procedure code).

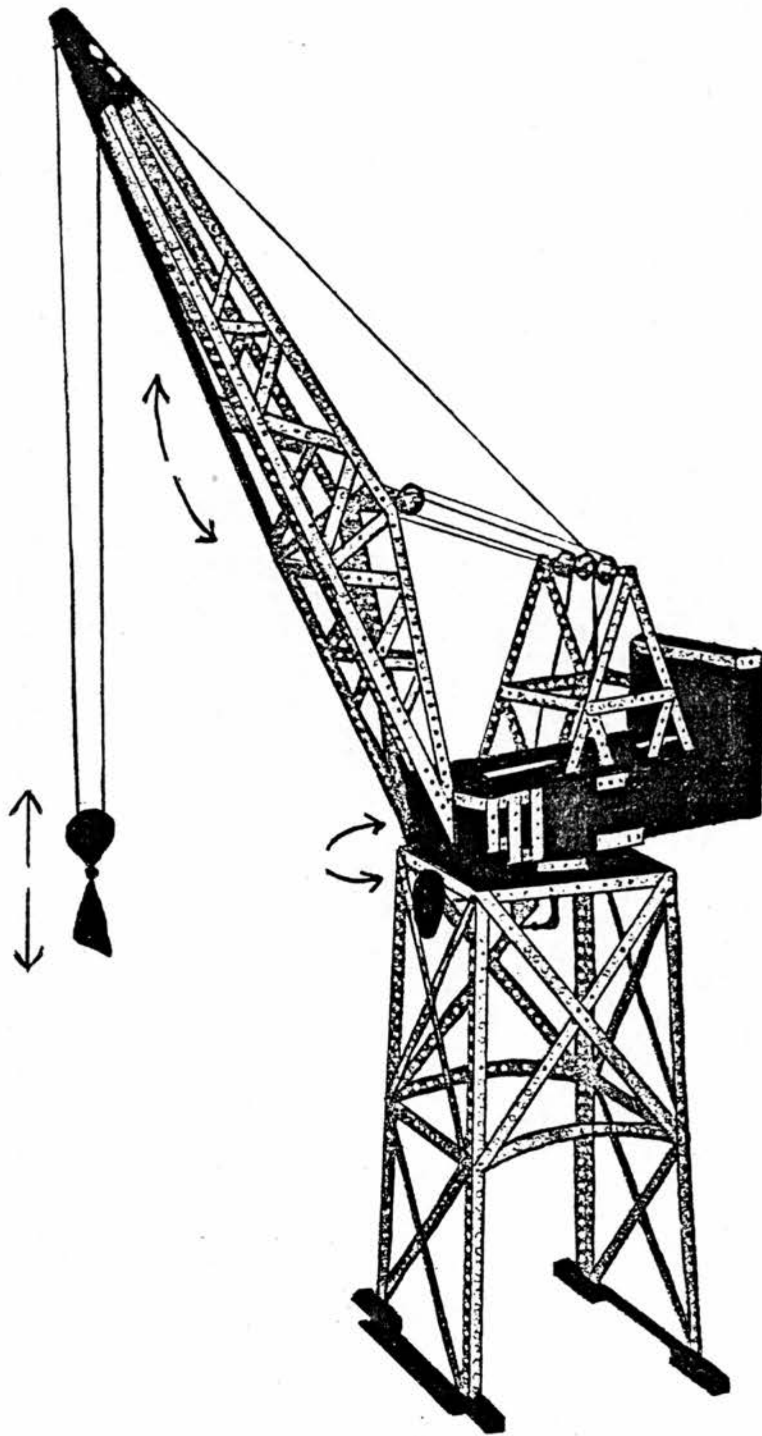


Figure 2 Crane

APPENDIX IV

CONTROL APPLICATIONS TESTS

PRE-TEST

Name:
Date:

This test is to find out something about your experience and knowledge about control applications. It does not matter if you find the questions difficult. Even if you do, please try to answer all the questions and to give the fullest explanation. Please put your answers on the blank papers provided. You can take as long as you like over the test.

(1) (a) Do you like building working models of any kind?

(b) What have you built before?

(c) What did you use for building - e.g. Meccano, Lego, wood?

(2) (a) Have you done any programming before?

(b) Which programming languages have you used?

(c) For each of the following, explain what they mean and why they are important features of a programming language for writing control programs:

(i) input,
(ii) output,
(iii) procedure,
(iv) conditional execution, and
(v) parallel processing?

(3) (a) Are you familiar with electronics?

(b) What is a sensor? Please give examples.

(c) What is an actuator? Please give examples.

(4) (a) Have you any idea how a robot arm is programmed to do a sequence of actions?

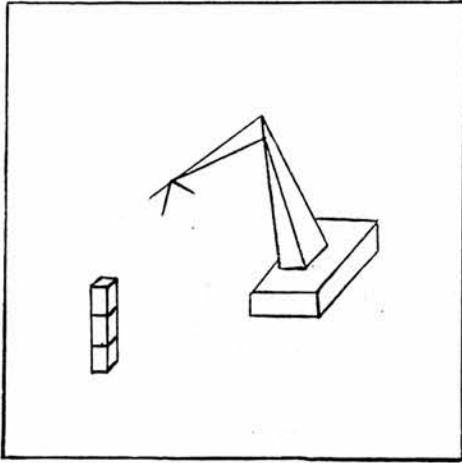
If so, please describe.

(5) What is feedback.

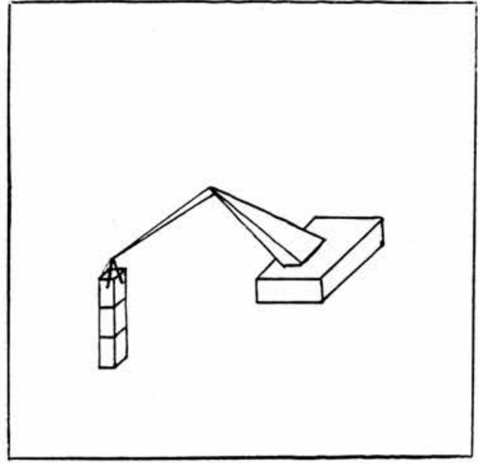
(6) Figure 1 shows a sequence of positions of a robot arm un-stacking the top block of a pile of blocks. The robot arm does not have touch sensors or cameras attached to it.

(a) Which of the positions shown must the computer remember in order to repeat the sequence of un-stacking the top block?

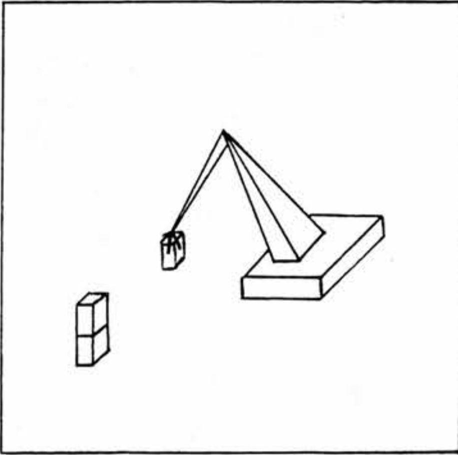
(b) Is there any crucial position missing from the figure? If so, illustrate it with a simple diagram.



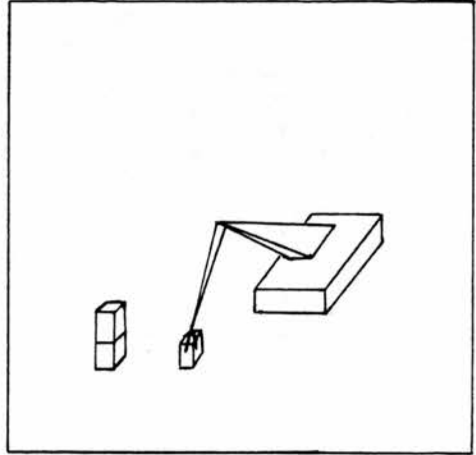
Position 1



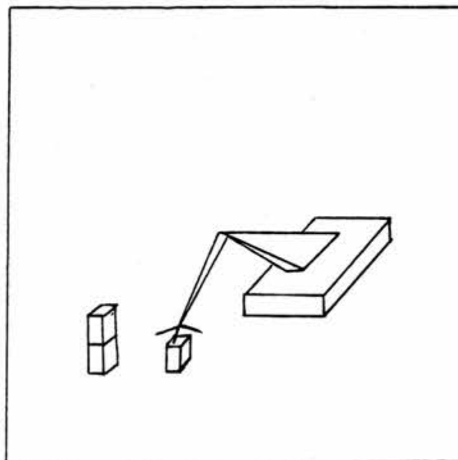
Position 2



Position 3



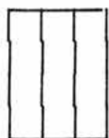
Position 4



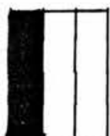
Position 5

Figure 1

(7) The letters A to I are represented by the following patterns:



A B C D



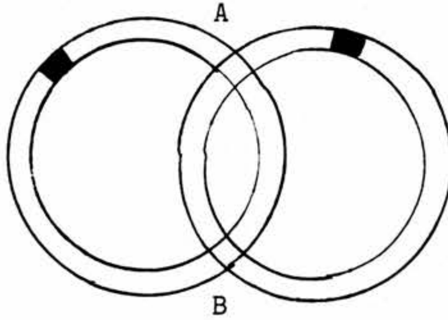
E F G H

Each pattern has three bars. Each bar is either black or white. We shall call these patterns bar codes.

(a) Using the given information above, what does the following sequence of bar codes represent:

(b) Describe a control device that can recognise this kind of bar coded information.

(8) The following diagram shows two railway tracks with a locomotive on each one of them.



The locomotives are computer controlled, i.e. the computer can make them move forward and stop.

How would you program these locomotives so that they can move along their own tracks without crashing into each other, either at junction A or B? You can make use of extra electronic components if you like.

POST-TEST

Name:
Date:

This test is to find out how much you have learned from the course It does not matter if you find the questions difficult. Please still try to answer all the questions and to give the fullest explanation. For a multiple choice question please put a circle around the most appropriate answer. For other questions, please put your answers on the blank papers provided. You can take as long as you like over the test.

(1) How is a reed switch activated?

(a) When it is immersed in water.
(b) When it is placed close to a magnet.
(c) When it is placed just above a white surface.
(d) When the temperature is in a particular range.
(e) When it is being pressed.

(2) How is a reflective-opto switch activated?

(a) When it is immersed in water.
(b) When it is placed close to a magnet.
(c) When it is placed just above a white surface.
(d) When the temperature is in a particular range.
(e) When it is being pressed.

(3) What is the main difference between working with DC motors and stepping motors?

(a) DC motors turn faster than stepping motors.
(b) DC motors consume less power than stepping motors.
(c) Stepping motors can only turn in one direction but DC motors can turn in either direction.
(d) It is easier to control how far a stepping motor turns than how far a DC motor turns.
(e) Stepping motors can lift heavier weight than DC motors.

(4) If you can switch a DC motor on and off by issuing commands to a computer, how can you control the speed of the motor?

(a) By switching the motor on and off and altering the duration that the motor stays off.
(b) By altering the work load of the computer, i.e. give the computer lots of things to do to reduce the speed of the motor, and give it less work to do to increase the speed of the motor.
(c) the speed of a DC motor can not be altered by computer commands.

(5) For each of the following, explain what they mean and why they are important features of a programming language for writing control programs:

(i) input,
(ii) output,
(iii) procedure,
(iv) conditional execution, and
(v) parallel processing

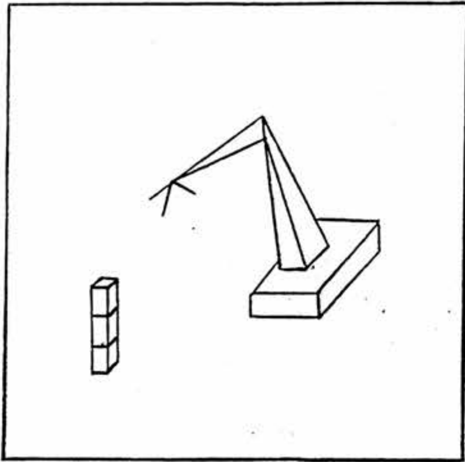
(6) Describe the 'teaching by showing' method of programming a robot arm to do a sequence of actions.

Describe the limitations of this method.

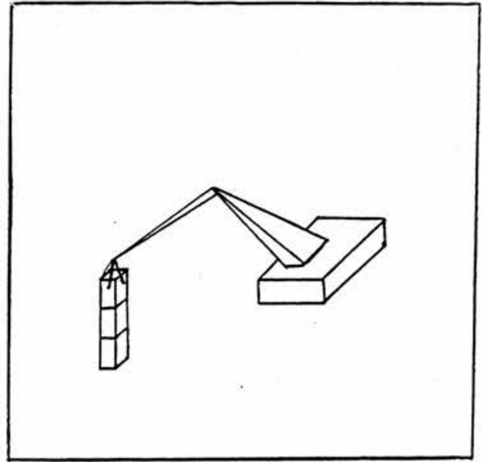
(7) Figure 1 shows a sequence of positions of a robot arm un-stacking the top block of a pile of blocks. The robot arm does not have touch sensors or cameras attached to it.

(a) Which of the positions shown must the computer remember in order to repeat the sequence of un-stacking the top block?

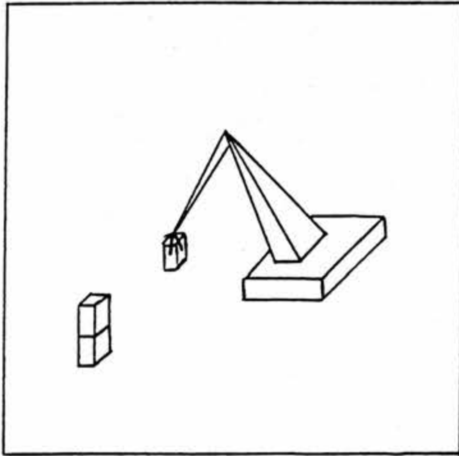
(b) Is there any crucial position missing from the figure? If so, illustrate it with a simple diagram.



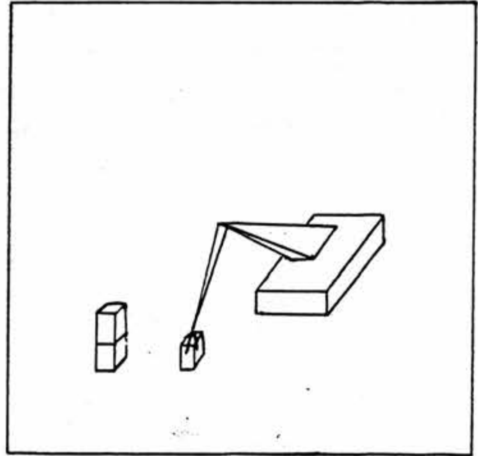
Position 1



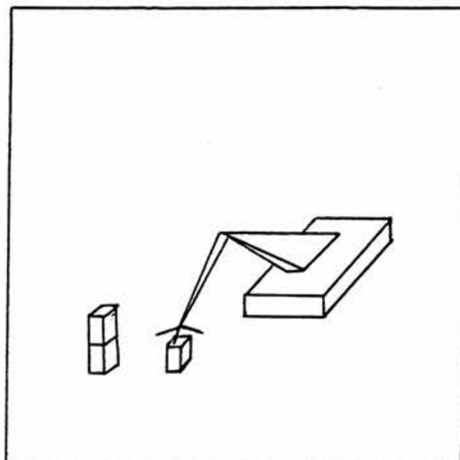
Position 2



Position 3



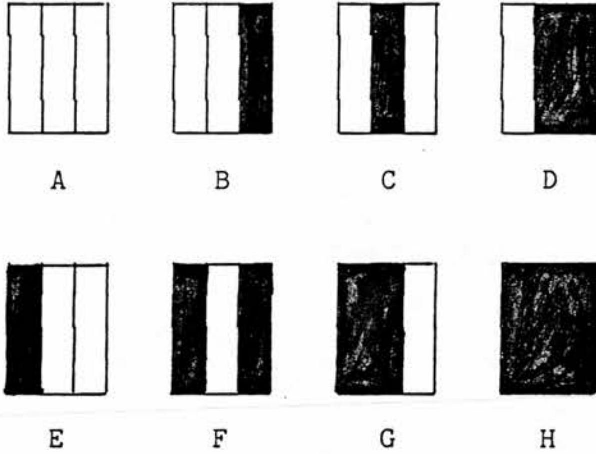
Position 4



Position 5

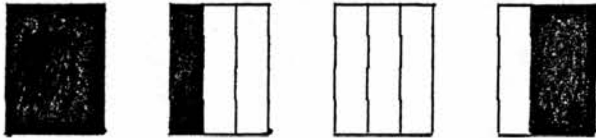
Figure 1

(7) The letters A to I are represented by the following patterns:



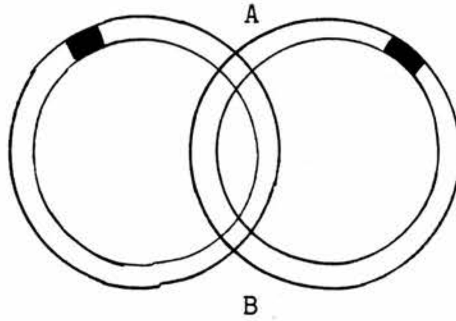
Each pattern has three bars. Each bar is either black or white. We shall call these patterns bar codes.

(a) Using the given information above, what does the following sequence of bar codes represent:



(b) Describe a control device that can recognise this kind of bar coded information.

(9) The following diagram shows two railway tracks with a locomotive on each one of them.



The locomotives are computer controlled, i.e. the computer can make them move forward and stop.

How would you program these locomotives so that they can move along their own tracks without crashing into each other, either at junction A or B? You can make use of extra electronic components if you like.

REFERENCES

Abelson, H. (1980) "Logo for the Apple II". Peterborough, NH: Byte/McGraw-Hill.

Abelson, H. and DiSessa, A. (1981) "Turtle Geometry: the computer as a medium for exploring mathematics". Cambridge, Mass: MIT Press.

Aabelson, H. and Goldberg, P. (1977) "Teacher's guide for computational models of animal behaviour". Logo Memo No. 46, The Artificial Intelligence Laboratory, Massachusetts Institute of Technology.

Aho, A.V. and Ullman, J.D. (1978) "Principles of Compiler Design". Addison Wesley.

Andrews, P.J. and Whittome, L.J. (1981) "Constructing Control Hardware". The Advisory Unit for Computer Based Education, Hatfield, Herts.

Arblaster, A. (1982) "Human factors in the design and use of computing languages". Int. J. Man-Machine Stud., 17, 211-224.

Atherton, R. (1982) "Structured Programming with COMAL". Chichester: Ellis Horwood Limited.

Avis, P. and Else, K. (1981) "Computer Controlled Railway". Computer Education, June, 5-6.

Bamberger, J. (1972) "Developing a musical ear: a new experiment". A.I. Memo No. 264, The Artificial Intelligence Laboratory, Massachusetts Institute of Technology.

Bamberger, J. (1979) "Logo music projects: experiments in musical perception and design". A.I. Memo No. 523, The Artificial Intelligence Laboratory, Massachusetts Institute of Technology.

Ben-Ari, M. (1982) "Principles of Concurrent Programming". Prentice-Hall International, Inc.

Bevis, G. (1984) "Microelectronics in schools and colleges". In 'Using Microcomputers in Schools', Terry (Ed.). Croom Helm Limited.

Bevis, G. and Trotter, M. (Eds.) (1981) "Microelectronics: Practical Approaches For Schools and Colledges". London: BP Educational Service.

Blackburn, P. (1980) "Monitoring experiments". In 'Micros in Schools', Volume 2, Case study No. 5.9, Open University.

Bobrow, D.G. and Raphael, B. (1974) "New programming languages for AI research". Computing, 6, 3, 153-174.

Bostock, M. (1983) "The BBC Buggy: an advanture with technology". Electronic Systems News, Autumn, 4-5. Hertfordshire: IEE.

Boden, M. (1977) "Artificial Intelligence and Natural Man". Hassocks: Harvester Press.

Borning, A. (1979) "Thinglab: a constraint-oriented simulation laboratory". Rep. No. 79-3, Xerox Palo Alto Research Center, California.

BP (1982) "BP Buildarobot Competition 1982-83". London: BP Oil Limited.

Brinch Hansen, P. (1975) "The programming language Concurrent Pascal". IEEE Trans. on Software Engineering, 1, 2, 199-207.

Brown, P.T. (1979) "Writing Interactive Compillers and Interpreters". John Wiley and Sons.

Bundy, A. (1983) "What stories should we tell Prolog students?" Working Paper No. 156, Dept. of Artificial Intelligence, Edinburgh University.

Byrd, L. (1980) "Understanding the control flow of Prolog programs". In 'Proceedings of the Logic Programming Workshop'; Tarnlund (Ed.).

Canara, A.b. (1975) "Experiments in teaching children computer programming". Technical Report No. 271, Institute for Mathematical Studies in the Social Sciences, Standford University, California.

Chung, W.H. (1984) "Terak Concurrent-Logo manual". Occasional Paper No. 49, Dept. of Artificial Intelligence, Edinburgh University.

Clark, K.L. and McCabe, F.G. (1982) "The control facilities of IC-Prolog". In 'Expert Systems in the Micro Electronic Age', Michie (Ed.). Edinburgh: University Press.

Clark, K.L. and Gregory, S. (1983) "PARLOG: a parallel logic programming language". Research Report DOC 83/5, Dept. of Computing, Imperial College, London University.

Clocksin, W.F. (1985) "Design and simulation of a sequential Prolog machine". New Generation Computing, 3, 101-120. Springer-Verlag.

Coll, J. (1982) "The BBC Microcomputer User Guide". London: British Broadcasting Corporation.

De Grandis-Harrison, R, (1983) "Forth on the BBC microcomputer". Cambridge: Acornsoft Limited.

DES (1981) "Microelectronics Education Programme: The Strategy". London: Department of Education and Science.

DES (1985) "Microcomputers in secondary schools: a survey of England, Wales and Northen Ireland secondary schools". London: BBC Educational Broadcasting Services Research Unit.

Dijkstra, E.W. (1968) "Cooperating sequential processes in programming languages". In 'Programming languages', Genuys (Ed.). New York: Academic Press.

Dijkstra, E.W. (1972) "The humble programmer". CACM, 15, 859-866.

Dijkstra, E.W. (1982) "How do we tell truth that might hurt?" ACM SIGPLAN Notices, 17, 5, 13-15.

Di Sessa, A. (1980) "Computation as a physical and intellectual environment for learning physics". Computers and Education, 4, 66-75.

du Boulay, J.B.H. (1978) "Learning primary mathematics through computer programming". Ph.D. Thesis, Dept. of Artificial Intelligence, University of Edinburgh.

du Boulay, J.B.H., O'Shea, T. and Monk, J. (1981) "The black box inside the glass box: presenting computing concepts to novices". Int. J. Man-Machine Stud., 14, 237-49.

Ennals, J.R. (1982) "Beginning micro-Prolog". Chichester: Ellis Horwood and Heinemann Computers in Education.

Ennals, J.R. (1984) "Teaching logic as a computer language in schools". In 'New Horizons in Educational Computing', Yazdani (Ed.). Chichester: Ellis Horwood.

Fisher, D.A. (1972) "A survey of control structures in programming languages". ACM SIGPLAN Notices, 7, 2, 1-14.

Feurzeig, W., Papert, S., Bloom, M., Grant, R. and Solomon, C. (1969) "Programming language as a conceptual framework for teaching mathematics". Report No. 1899, Bolt Beranek and Newman Inc., Cambridge, Mass.

Feurzeig, W., Horwitz, P. and Nickerson, R.S. (1981) "Microcomputers in education". Report No. 4798, Bolt Beranek and Newman Inc., Cambridge, Mass.

Foster, C.C. (1981) "Real Time Programming - Neglected Topics". Massachuetts: Addison-Wesley Publishing Company.

Ginn, A. (1984) "Giving maths power to the children". Practical Robotics, Nov, 29-33.

Goldberg, A. (1977) "Smalltalk in the classroom". Rep. No. SSL 77-2, Xerox Palo Alto Research Centre, California.

Goldberg, A. and Ross, J. (1981) "Is the Smalltalk-80 system for children?" Byte, 8, 6, 347-68.

Goos, G. and Hartmanis, J. (Eds.) (1983) "Reference Manual For The Ada Programming Language". Springer-Verlag.

Gould, L. and Finzer, W. (1981) "A study of TRIP: a computer system for animating time-rate-distance problems". In 'Computers in Education', Lewis and Tagg (Eds.). Amsterdam: North-Holland.

Grant, R. (1980) "GEIGER". In 'Micros in Schools', Volume 2, Case Studies No. 5.3, Open University.

Habermann, A.N. and Nassi, I. R. (1980) "Efficient implementation of Ada Tasks". Computer Science Report CMU-CS-80-103, Carnegie Melon University.

Hardy J. and Hardy M. (1985) "Some desirable improvements in the Logo language". Paper presented at the British Logo User Group 1985 annual conference. To be published in Logo Almanack Vol 3.

Hartley, J.R. (1980) "Using the computer to assist the learning of mathematics". In 'Proc. of British Society of the Psychology of Learning Mathematics Conference', Nottingham University.

Harvey, B. (1984) "Why Logo?" In 'New Horizons in Educational Computing', Yazdani (Ed.). Chichester: Ellis Horwood.

Higgs, J.C. (1980) "The Leicestershire Schools Engineering Project - A schools/industry joint venture". Fishers Controls Ltd.

Hoare, C.A.R. (1974) "Monitors: an operating system structuring concept". CACM, 17, 10, 549-557.

Hoare, C.A.R. (1983) "Hints on programming language design". In 'Programming Languages A Grand Tour', Horowitz (Ed.). New York: Springer-Verlag.

Horowitz, E. (1983) "Fundamentals of Programming Languages". New York: Springer-Verlag.

Howe, J.A.M. (1978) "Artificial Intelligence and computer-assisted learning: ten years on". PLET, 15, 2, 114-125.

Howe, J.A.M. (1980) "Learning through model building". In 'Expert Systems in the Micro Electronic Age', Michie (Ed.). Edinburgh: University Press.

Howe, J.A.M. and Delamont (1974) "Towards an evaluation strategy for CAI projects". Bionics Research Reports: No. 15, School of Artificial Intelligence, University of Edinburgh.

Howe, J.A.M. and du Boulay, B. (1979) "Microprocessor assisted learning: turning the clock back?" PLET, 16, 3.

Howe, J.A.M., O'Shea, T. and Plane, F. (1980) "Teaching mathematics through Logo programming: an evaluation study". In 'Computer Assisted Learning: Scope Progress and Limits', Lewis and Tagg (Eds.). Amsterdam: North Holland.

Howe, J.A.M., Ross, P.M., Johnson, K.R., Plane, F. and Inglis, R. (1982) "Learning mathematics through programming: the transition from laboratory to classroom". Working Paper No: 118(b), Department of Artificial Intelligence, Edinburgh University.

Hooper, R. (1977) "The National Development Programme in Computer Assisted Learning: Final Report of the Director". London: Council for Ed. Tech.

Howard, C. and Hooton M. (1981) "The House Project". Electronics Systems News, July, 10-11. Hertfordshire: IEE.

Johnson, K. (1983) "The hitch hicker's guide to LOGO". Oxford: Research Machines Ltd.

Johnson, R., Procter, C. and Reglinski, A. (1984) "Interfacing and Control on the BBC Micro". England: National Extension College Trust Limited.

Kay, A. (1977) "Microelectronics and the personal computer". Scientific America, 237, 3, 230-224.

Kelman, P (1983) "Seymour, has your dream come true?" Classroom computer News, No. 5, 7.

Kowalski, R.A. (1974) "Predicate logic as programming language". In 'Proc. IFIP-74 Congress'. Amsterdam: North-Holland, 569-574.

Kowalski, R.A. (1984) "Logic as a computer language for children". In 'New Horizons in Educational Computing', Yazdani (Ed.). Chichester: Ellis Horwood.

Kurland, D.M. and Pea, R.D. (1983) "Children's mental models of recursive Logo programs". Technical Report No. 10, Center for Children and Technology, Bank Street College of Education, New York.

Lawler, B. (1984) "Designing computer-based microworlds". In 'New Horizons in Educational Computing', Yazdani (Ed.). Chichester: Ellis Horwood.

McCabe, F.G. (1980-81) "Micro-PROLOG Programmer's Manual". London: Logic Programming Associates Ltd.

Mellish, C.S. (1982) "An alternative to structure sharing in the implementation of a Prolog interpreter". In 'Logic Programming', Clark and Tarnlund (Eds.). Academic Press.

Mickel, A.D. (1981) "The future of Pascal". In 'Pascal - The language and its implementation', Barron (Ed.). John Wiley and Sons Ltd.

Milner, S. (1973) "The effects of computer programming on performance in mathematics". E.R.I.C. Report No. ED076391.

Motiwalla, S. (1982) "Development of a software system for industrial robots". Mechanical Engineering, August, 36-39.

Musha, D.R. (1981) "TI Logo Manual". Texas Instruments Incorporated.

Naish, L. (1982) "An introduction to MU-Prolog". Technical Report 82/2, Department of Computer Science, Melbourne University.

Naish, L. (1983) "Automatic generation of control for logic programs". Technical Report 83/6, Department of Computer Science, Melbourne University.

Noss, R. (1983) "Doing maths while learning Logo". Mathematics Teaching, 104, September.

O'Shea, T. and Self, J (1983) "Learning and Teaching with Computers - Artificial Intelligence in Education". Sussex: The Harvester Press Ltd.

Papert, S. (1971a) "A computer laboratory for elementary schools". A.I. Memo No. 246, The Artificial Intelligence Laboratory, Massachusetts Institute of Technology.

Papert, S. (1971b) "Teaching children thinking". A.I. Memo No. 247, The Artificial Intelligence Laboratory, Massachusetts Institute of Technology.

Papert, S. (1980) "Mindstorms: Children, computers and powerful ideas". New York: Basic Books, Inc.

Papert, S., Watt, D., Di Sessa, A. and Weir, S. (1979) "Final report of the Brookline Logo Project, Part II: project summary and data analysis". A.I. Memo No. 545, The Artificial Intelligence Laboratory, Massachusetts Institute of Technology.

Parlett, M. and Hamilton, D. (1977) "Evaluation as illumination: a new approach to the study of innovatory programs". In 'Beyond the numbers game', Hamilton , Jenkins, King, MacDonald and Parlett (Eds.). Macmillan Education Limited.

Pea, R.D. (1983) "Logo programming and problem solving". Technical Report No. 12, Center for Children and Technology, Bank Street College of Education, New York.

Pea, R.D. & Hawkins, J. (1983) "A microgenetic study of planning processes in a chore-scheduling task". In 'Blueprints for thinking: The development of social and cognitive planning skills', Friedman, Scholnick, and Cocking (Eds.). New York: Cambridge University Press.

Pea, R.D. & Kurland, D.M. (1983) "Logo programming and the development of planning skills". Technical Report No. 16, Center for Children and Technology, Bank Street College Education, New York.

Pike, T.D. (1982) "The Schools Council modular courses in technology development of a unit entitled 'Microelectronics in control'". Electronics Systems News, May, 17-18. Hertfordshire: IEE.

Ripley, G.D. and Druseikis, F.C. (1978) "A statistical analysis of syntax errors". Computer Languages, 3, 227-240.

Ross, P. and Howe, J.A.M. (1984) "The design of Edinburgh Logo". Microprocessors and Microsystems, 8, 3.

Rowe, N. (1976) "Grammer as a programming language". A.I. Memo No. 391, The Artificial Intelligence Laboratory, Massachusetts Institute of Technology.

Sammet, J. (1969) "Programming Languages: History and Fundamentals". Prentice-Hall.

Schools Committee Working Party (1980) "Syllabuses for the future". London: British Computer Society.

Sergot, M. (1984) "A query-the-user facility for logic programming". In 'New Horizons in Educational Computing', Yazdani (Ed.). Chichester: Ellis Horwood.

Shapiro, E.Y. (1983) "A subset of Concurrent Prolog and its interpreter". Technical Report TR-003, ICOT, Japan.

Sharples, M. (1980) "A computer based language workshop". ACM SIGCUE Bulletin, 14, 3.

Sime, M.E., Green, T.R.G. and Guest, D.J. (1977) "Scope marking in computer conditionals - a psychological evaluation". Int. J. Man-Machine Stud., 9, 107-118.

Simmonds, K. (1982) "Control technology a place within the curriculum". Electronics Systems News, May, 16-17. Hertfordshire: IEE.

Sleeman, D. and Brown, J.S. (Eds.) (1982), "Intelligent Tutoring Systems". London: Academic Press.

Solomon C. (1982) "Introducing Logo to children". Byte, 7,8.

Soloway, E. and Ehrlich, K. (1982) "What do novices know about programming?". In 'Directions in human/computer interaction", Badre and Shneiderman (Eds.). New Jersey: Ablex Publishing Corporation.

Sparkes, R.A. (1982) "Microcomputers in science teaching". School Science Review, 63, 224, 442-452.

Stallman, R.M. (1985) "GNU Emacs Manual". Publication Department, Massachusetts Institute of Technology.

Statz, J. (1973) "Syracuse University Logo Project: Final Report". Syracuse University, New York.

Stevenson, P. (1980) "Computers and control in schools". In 'Proceedings of Microcomputers in Education Seminar'. Northwood Hills: Online Conferences Limited.

Tennent, R.D. (1981) "Principles of Programming Languages". Prentice-Hall.

Thomson, P., Bromley, S. and Higgins, J. (1984) "BP Buildarobot Competition 1983". Computer Education, June, 2-5.

Warren, D. (1983) "An abstract Prolog instruction set". Technical Note 309, Computer Science and Technology Division, SRI International.

Weir, D.J. (1982) "Teaching logic programming: an interactive approach". M.Sc. Thesis, Dept. of Computing, Imperial College London.

Weston, P. (1984) "Teaching about computing". In 'Using Microcomputers in Schools', Terry (Ed.). Croom Helm Limited.

Weyer, S.A. and Cannara, A.B. (1975) "Children learning computer programming: experiments with languages, curricula and programming devices". Technical Report No. 250, Institute for mathematical studies in the social sciences, Standford University.

Wilson, R.J. (1984) "Machine code programming in the physics laboratory". Computer Education, June, 6-8.

Winston, P.H. and Horn, B.H. (1981) "LISP". Massachusetts: Addison-Wesley Publishing Company.

Wirth, N. (1974) "On the design of programming languages". In 'Proc. IFIP Congress 74', Amsterdam:North-Holland, 386-393.

Wirth, N. (1977) "Modula: a language for modular multiprogramming". Software Practice and Experience, 7, 3-35.

Wood, J. (1981) "Using Control Hardware: For Computer Studies Courses". Advisory Unit For Computer Based Education, Hatfield, Herts.

Yazdani, M. (Ed.) (1984) "New Horizons in Educational Computing". Chichester: Ellis Horwood.

Young, S.J. (1982) "Real Time Languages". Chichester: Ellis Horwood Limited.