

THE UNIVERSITY OF CALGARY

TOPICS AND TOOLS IN THE INTRODUCTORY COMPUTER SCIENCE
CURRICULUM

by
KATRIN BECKER

A THESIS
SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

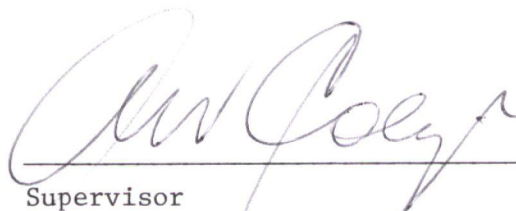
DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA
AUGUST, 1983


© KATRIN BECKER 1983

THE UNIVERSITY OF CALGARY
FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled, "Topics and Tools in the Introductory Computer Science Curriculum" submitted by Katrin Becker in partial fulfillment of the requirements for the degree of Master of Science.



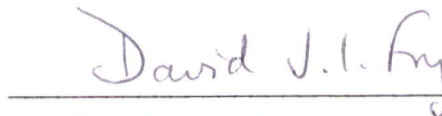
Supervisor
Dr. Anton W. Colijn
Department of Computer Science



Dr. Michael R. Williams
Department of Computer Science



Dr. John Slater
Malaspina College



Dr. David J.I. Fry
Department of Physics

ABSTRACT

The research outlined in this thesis deals primarily with the introductory computer science curriculum (i.e. the first two years). The topics and objectives are outlined first. This is the body of information that a finishing second year student should be expected to know. The results of the questionnaire given to computer science department members are discussed and then tools used for teaching computer science, both current and future, are described.

ACKNOWLEDGEMENTS

I am indebted to my supervisor, Dr. Anton Colijn, whose advice and guidance helped keep me on track and in school. I also wish to thank my former Dr. John Slater for his many good ideas and for convincing me that the work was worthwhile, as well as for taking the time to see it through to the end.

Thanks also to the other two members of my committee, Dr. David Fry for his participation, and to Dr. Mike Williams for his time and concern, not only at the defense, but throughout the years I have spent at the University.

Although too numerous to mention them all by name, I wish to thank all those who answered my questionnaire, and took the time to talk to me, as well as those members of this department who let me use their classes.

Finally, I thank my mother, Renate Bischof, for all her help and support, and my mother-in-law, Vivian Parker, who typeset this thesis, and Jim Parker, without whom I could not have made it this far.

TABLE OF CONTENTS

Abstract	iii
Acknowledgements	iv
List of Tables and Figures	vi
Chapter	Page
1 Introduction	1
2 Topics and Objectives	4
3 Interrelationships	9
4 Questionnaire Results	18
5 Recommendations	35
6 Current Tools	42
6.1 Software Tools	46
6.2 Hardware Tools	49
6.3 Other Tools	50
7 A Few Additions	54
7.1 Syntax Diagrams	54
7.2 Microtool	57
7.3 Finite State Machine Simulator	58
8 New Tools	61
8.1 Recursion	61
8.2 Floating Point Demonstrator	62
8.3 Treetool	63
9 Conclusions	68
References	70
Textbooks	73
Appendix	75

LIST OF TABLES AND FIGURES

Table	Page
4.1 Tools/Techniques	33
5.1 Areas of Need	38
8.1 Treetool — Commands	65
8.2 Treetool — Language Definitions	66
 Figure	 Page
2.1 Major Categories	6
3.1 Very Strong Interdependence of Topics	13
3.2 Strong Interdependence of Topics	14
3.3 Some Interdependence of Topics	15
3.4 Final Structure — Interdependence of Topics	16
4.1 University of Victoria — Course Plan	19
4.2 University of British Columbia — Course Plan	20
4.3 Simon Fraser University — Course Plan	21
4.4 University of Alberta — Course Plan	22
4.5 University of Calgary — Course Plan	23
4.6 University of Lethbridge — Course Plan	24
4.7 University of Saskatchewan — Course Plan	25
4.8 University of Regina — Course Plan	26
4.9 Topics Covered in the Introductory Curriculum	27
4.10 Degree of Coverage	29
4.11 Satisfaction with Results	29
6.1 Distribution of Tools Used	43
6.2 SP/k Subsets	46
7.1 Example Exercise 1	55
7.2 Example Exercise 2	56
7.3 Commands for Finite State Machine Simulator	59

Chapter 1

INTRODUCTION

Computer science is the study of solving problems in information manipulation. Computer science has existed as a discipline (taught in universities) for approximately twenty years — not very long in comparison to other disciplines, some of which have been studied for centuries. In most universities, computer science had developed as part of another field, most often mathematics, but also engineering and sometimes business. Since their inception, most computer science sections of these departments have become departments in their own right.

In 1968, the ACM produced their first curriculum [C3S 68]. In this document, the ACM Curriculum Committee outlined its recommendations for undergraduate programs. A classification for subject areas was presented and various courses in those areas were described.

Nine years later, the IEEE Computer Society published its own curriculum. Even though it was intended for computer science and engineering [IEEE77], it concentrates its efforts most heavily on the teaching of engineering and as such does not have a great deal of impact on computer science.

By 1978, the ACM has produced its second computer science curriculum [Aust78]. It provides an extensive list of topics considered necessary to students of computer science. The committee singled out those topics deemed to be part of the elementary curriculum. This list forms the basis for the list given in the appendix of this thesis. Since the initial curriculum in 1968, numerous other groups have endeavoured to define the computer science curriculum [Coug73, Muld75, Engl78]. Clearly, the discipline is maturing. The time has come to define fully the discipline and develop a strategy for teaching it.

Although many have tried to list the required topics in this field, few have actually defined, or given objectives for the topics they list. Most describe overall objectives, but few go as far as stating what the abilities of the students should be when they

have mastered each topic. This is one of the major goals of this thesis.

Outlining the objectives of any given subject area gives a direction for teaching and a means of identifying achievement (a way of knowing when the student has succeeded). Clear objectives also allow the instructors to keep in touch with the task at hand; they allow the instructor to plan and present an organized course that instructors in subsequent courses can build on with confidence. The importance of this, from the students' point of view, is that they know what is expected of them. This allows them to prepare for the course and it also gives the students a sense of direction — they too will be able to identify success. It should be noted however, that this type of approach is really only possible when dealing with material that has primarily a skill oriented or knowledge based component. Judging success is far more difficult when dealing with a deep understanding or philosophical problem. It is still possible to outline objectives but determining success is not quite so clear cut.

The subject of this thesis is very closely related to the study of education, and education theory plays a prominent role, but it is felt very strongly that the research outlined in this document could not be completed by someone schooled solely in education. Such a person would know much of the theory of education and approaches to solving the problems experienced by computer science departments, but they would lack the familiarity with the subject matter required to make valid decisions about required topics and their interrelationships.

With the rapidly increasing student numbers and escalating demands placed upon all computer science faculty, it is obvious there is a need for a clear definition of the subject matter and the means for making the task of teaching it somewhat less demanding of both time and energy.

Using the ACM curriculum as the primary source, a list of topics to be required of elementary computer science students was created. With the aid of dictionaries and textbooks, objectives were added to the topics. This resultant list, given in the appendix is discussed in the second chapter. Chapter 3 attempts to describe how these topics are interrelated. By listing the interdependencies, it is possible to decide on a rough order and outline of courses to be offered in such a curriculum.

In order to come up with an accurate list of topics and objectives covered in the elementary curriculum, several Western Canadian universities were interviewed and the

faculty presented with questionnaires. The results of these questionnaires are described in chapter 4. From these results it was possible in chapter 5, to outline the areas of need and describe tools that could be used to fulfill these needs.

It is quite alarming that departments of computer science do not use the computer as a tool (writing programs excepted). For various reasons, little development of tools for the purposes of teaching computer science has taken place. Chapter 6 introduces some of the tools that are currently available. Several new tools have been designed and implemented. These are described in chapter 7, along with preliminary results from their testing (where available). Chapter 8 proposes several additional tools, not yet implemented, that fulfill several of the areas of need, and the conclusions appear in chapter 9.

Chapter 2

TOPICS AND OBJECTIVES

One of the major goals of this thesis is to research, identify, and describe the major concepts of importance in the elementary computer science curriculum. This chapter describes the list (found in the appendix) that resulted from this research. The major sections are described in turn and some topics that warrant special mention are discussed in greater detail.

The initial list of topics was drawn up using the ACM curriculum guide [Aust79] and various introductory textbooks (see list following references) as well as through consultation with various individuals [Slat82, Park83a]. This resulted in a very long list of topics, many of which overlapped with each other. Quite often many different terms are used to refer to very similar topics.

In an attempt to eliminate this overlap, several dictionaries were used (including 2 computer science dictionaries [Sipp76, Chan77]) as well as a thesaurus. Smaller topics with a common core were grouped into somewhat larger concept blocks (for example, definition, call and expansion of macros in assembler were combined with assembly time computation, conditional assembly and parameter handling to form the block called 'macros'.)

As explained earlier (Chap. 1) a set of objectives was drawn up for each topic describing what an elementary computer science student should be able to do once that topic has been mastered. It is very important to set goals with each topic to provide direction. To make this somewhat clearer, consider the following example. Most would agree that every student in an elementary curriculum should understand the common parameter passing techniques. This is a fairly broad subject though, and outlining what the student is expected to know at the elementary level is quite important. There is a substantial difference between being able to trace the execution of a program that uses parameters by hand and predicting the result, and being able to implement these parameter passing mechanisms. Very few would expect their elementary students to do the latter, but this information is not revealed by looking

only at the topic heading.

Guided largely by past experience and the course descriptions from various Western Canadian universities, these topics were then placed into 12 major categories (see Figure 2.1). One of the primary reasons for the higher level division was a desire to create a common thread for a large group of concept blocks. A secondary reason but perhaps almost as important was to break up the list into manageable chunks. Even with a guide, some of the choices were of necessity somewhat arbitrary, and one could no doubt argue that a particular topic was more closely related to one category than another. Quite often in such cases there are as many arguments for having the topic in one section as there are for having it in the other. For example, although few would suggest that a topic such as number representation is a topic of theory, it is more difficult to draw the line between the architecture and logic design aspects of this topic. It is found here (in the appendix) in the logic design section for two reasons: it follows quite nicely from sequential circuits, which is much more a topic of logic design than architecture, and the other reason is one of practicality: that is, the architecture section is already quite large.

One of the major difficulties encountered while creating this list of topics was one of terminology. As it is still a relatively young discipline, many terms were required that did not previously exist in the context of computer science. In consequence, each group of researchers came up with their own terms as they were required. The result is that often several names exist for very similar ideas yet each has subtle differences so they all remain in use. Because of this problem terms have been used in some parts of the list with which many may not be familiar. These will now be explained, along with a more detailed explanation of the choice of categories.

'Introduction to Computers' includes all of the information that is typically covered in the first few lectures of an introductory computer science course. It is intended to introduce the students to the idea of a computer, its major parts, and its operation.

'Operational Use of the Computer' is an area that is quite often covered in labs or tutorials, usually by a teaching assistant. This section serves to provide the students with a working knowledge of the computer they will be using as well as the necessary skills to complete their assignments and exercises. One topic in this section that deserves special mention is that of keyboard skills. Proficiency at the use of the

Figure 2.1 Major Categories

1. Introduction to Computers
2. Operational Use of the Computer
3. Programming Techniques
4. Programming Language Structure
5. Logic Design
6. Architecture
7. Discrete Mathematics
8. Theory of Computation
9. Data Structures
10. Storage Management
11. Other Major Topics
12. Miscellaneous

keyboard (typing) is a skill that every computer scientist needs, but also one that very few get the opportunity to learn formally. With so many new ideas to learn at once in a first course, it is unfortunate that students are forced to add to their frustration by having to fumble with an unfamiliar keyboard at the same time.

The next two sections are very closely related, and in fact could be combined under the main heading of 'Elementary Software Design'. It has been split into two sections to show the two different aspects of the art of programming: developing an algorithm and turning it into a program ('Programming Techniques'), and the structure of the program itself (i.e. how the statements are executed and the syntax and semantics of the language), covered in 'Programming Language Structure'.

The concept of abstract data structures is a useful one (in 'Programming Techniques'), even if the students do not have access to a language that implements them. Even though few universities include it in the elementary curriculum (see Chapter 4), many programmers have found it to be beneficial in helping them to organize their programs into well structured modules [Lisk74], and there is no reason to believe that this type of approach cannot do the same for students.

Some topics appear to be listed more than once. For example, parameters are referred to in several areas — in 'Programming Techniques', 'Programming Language

Structure', and in 'Architecture' — but in each case a different aspect of the topic is discussed. Under the heading of programming (in 'Programming Techniques'), parameters are mentioned in the context of program features, while in the programming language structure section, they are discussed in the context of scope, as well as the run-time environment. In the architecture section, where they are mentioned again, it is in connection with assembler language.

'Logic Design' is intended to include the lowest level of computer function of interest to a computer scientist, from the theory of digital circuits to digital arithmetic, and then on to an introduction to computer design. It fits in very closely with architecture and, in fact some topics can be argued as being part of architecture rather than logic design. For that matter, all of the material in both sections can be considered part of architecture, but in an effort to keep that section from becoming too large, it was broken up.

'Discrete Mathematics' requires little explanation other than the fact that since many areas of computer science rely heavily on concepts of discrete mathematics, it stands to reason that discrete mathematics should figure prominently in the elementary curriculum.

'Theory of Computation' uses the required background in abstract algebra to form a basis for understanding the nature and theoretical limitations of machines, programs, and programming problems. Although detailed study is usually left until the senior years, an early introduction to this area is helpful in several respects. It not only helps students understand their programs better, but also the machines on which these programs run. Another important benefit is that a knowledge of theory also helps develop the discipline of problem solving and the analytical approach required of all computer scientists.

The next section is 'Data Structures'. Although this is an area that could fall under the main heading of 'Software Design', it is itself of sufficient magnitude to warrant its own section. The primary objective at the elementary level should be a practical knowledge of the various data structures and their applications. A more detailed treatment can be left until the senior years. The same can be said of the section that deals with storage management.

The last two sections (Other Major Topics and Miscellaneous) of the list are not really

to be considered as part of the core, but it is felt that students entering the senior part of a computer science program should be familiar with the main areas of the field. as well as having had exposure to its history and present status in society.

The complete list is given in the appendix. It forms a description of the capabilities and knowledge that a student would ideally have acquired upon completion of the elementary part of a computer science program. It is not meant to detail completely the knowledge a computer scientist requires, but rather to form a solid foundation upon which to build.

Chapter 3

INTERRELATIONSHIPS

When discussing a curriculum in any discipline, simply listing the topics and stating the corresponding objectives does not reveal the complete picture. The mastery of one topic almost always depends on a prior understanding of some other topic or set of topics. In order to allow the students to absorb as much of the material presented to them as possible, it is very important that the sequence of presentation be carefully organized to provide maximum coherency [Good80].

As in categorizing the topics, there is a certain amount of repetition in describing their interdependencies. The discipline of computer science can be thought of as a 'Gestalt'. In this light, one could state that almost all of the subjects depend on information in all other areas to be complete (although the extent of the dependency varies). The ideal situation would be to cover all subjects simultaneously, but for practical reasons this is clearly impossible. Not only would this require so much time that the students could do nothing else, but the structure of each individual course would become so general that no-one would have sufficient depth of understanding in each subject to teach these courses effectively.

For the purposes of this discussion, every attempt has been made to outline only the most relevant relationships. Where applicable, the topics listed within each major category have been listed in chronological order, so for the most part, the way that topics in a category relate to each other need not be discussed further. There are, however, several exceptions.

Most of the topics under the third category ('Programming Techniques') must be dealt with in parallel, but the last few (software portability, software communication, numeric computations, manipulating character data, and recursion) can be covered separately in the most convenient manner. The same is true of section 6: 'Architecture'. Most of the topics must be dealt with simultaneously with the exception of the first few (Von Neumann machine and hardware systems organization) and the last few (microprogramming, etc.), which should be covered in the order indicated.

There are a few topics that stand out as separate from the rest. These will be dealt with individually. The first of these comes from the area of discrete mathematics. It is the topic of proofs. The importance of proofs is very much underrated in most computer science curricula, as indeed, it is in many disciplines [Poly57]. It is a topic seldom discussed in its own right and quite often one that students understand only superficially at best. The approach used in constructing a rigorous proof teaches a discipline that is applicable in all areas. As stated before, systematic problem solving is really what computer science is all about, and familiarity with the process of strict mathematical reasoning, as well as both formal and informal logic are clearly valuable assets.

Another topic of major importance also falls under the category of theory. If it is agreed that problem solving (along with the study of information) is the nucleus of computer science, then it follows that the analysis of algorithms should also be considered a central topic. The analysis of algorithms includes not only the study of computational, space, and time complexity and expected performance, but also various aspects of proving programs correct. Although one cannot possibly cover these topics in great detail within the context of the elementary curriculum, it is certainly possible to lay a solid foundation for further study. The major goals of this subject at this level should be to provide a good introduction and thereby make the student aware of this aspect of problem solving during their studies.

Information theory, although it probably fits most naturally in the category of 'Theory' bears very little relation to any of the other topics described in this section. It is however, along with proofs, and analysis of algorithms a fairly central topic. Knowledge of the basis of information theory (how codes are formed, how numbers and fractions are represented, etc.) help one to derive a more thorough understanding of various aspects of programming techniques (it helps to answer why certain programming errors occur as well as how one might approach various programming tasks in the light of how the information is actually stored and transmitted). It is obviously central to the understanding of data representation and various aspects of architecture. Information theory also plays an important role in the study of storage management.

There is one other topic that, although it fits in fairly well with the other topics of its group, deserves special mention due to its importance. Again, due to the nature of the

discipline (solving problems of information manipulation), sorting and searching, by definition, must play a very important role. It is vital that the students understand the major algorithms thoroughly to the point of being second nature (although not necessarily to this depth at the elementary level). Although sorting and searching are listed as topics separate from the others in the area of data structures, they cannot be dealt with as such, but instead should be discussed in connection with the specific data structures (sorting and searching arrays, lists, etc.).

Some other topics, although quite distinct and definable, cannot be taken as separate concept blocks. Clearly, it is not possible to discuss problem solving in one series of lectures, however long it may be, without discussing other topics (such as programming style, debugging, verification, etc.) along with it. The same holds true for many other topics. These must be covered in smaller sections over a period of time, going on to more advanced and involved aspects of the topics as the students' base of knowledge broadens (a 'concentric' approach).

The last two areas in the list of topics have not been included in the discussion of interrelationships to this point. Numerical methods and random numbers do not really fit with the other topics listed in section 11 ('Other Major Topics'), but even though numerical methods is a topic more closely related to programming techniques, it is one of sufficient magnitude to warrant being dealt with separately. Random numbers is also an important issue and one that elementary students should definitely be familiar with, but it does not lend itself to categorization under the major headings in this list.

The other topics in this section can be dealt with in any general computer science or programming course. They require almost no prior knowledge (except that found in section 1 and terminology from section 2) and as a result could be covered in a first course, if desired. Although students may be expected to be familiar with these areas and their application, little more need be done with them at the elementary level.

The miscellaneous section also requires little in the way of prior knowledge (especially with the first two — history and social issues) when covered more or less superficially. The last two, strictly speaking, also do not require much prior knowledge, but they are probably best left until the students have somewhat more knowledge of computer science in general so they may appreciate the material better.

The interrelationships shown in the following figures (3.1-3.3) were compiled using

the list in the appendix. Because of the complicated web of interrelationships the diagram showing how the topics relate to each other has been broken up into several parts. Figure 3.1 shows these areas that have a very strong dependence on other areas (almost all topics rely on knowledge gained from the study of almost all topics in the other area), thereby displaying a definite prerequisite and co-requisite structure (the source of the arrows denotes a prerequisite, and arrows pointing in both directions show areas that can be considered co-requisites). As the figure illustrates, an 'Introduction to Computers' (section 1) is required for every other area except 'Discrete Mathematics' and 'Theory'. 'Operational Use of the Computer' is also shown to be a prerequisite for almost every other area. What this figure does not show is that general terminology (from section 2) can be considered a prerequisite for every area, with the possible exception of 'Discrete Mathematics'. 'Programming Techniques' and 'Programming Language Structure' are co-requisites and fit together quite closely. Although neither are strong prerequisites for the data structures section, 'Discrete Mathematics' is. Both 'Data Structures' and 'Programming Techniques' serve as prerequisites for 'Storage Management'.

The next step in the development of the prerequisite structure is to illustrate those areas that have a strong dependence (although not complete) on each other. Figure 3.2 illustrates the interdependence at this level. Note the co-dependence of 'Programming Techniques' and 'Architecture', as well as the dependence of 'Storage Management' on an understanding of 'Programming Language Structure'.

The last stage exhibits a weaker dependence (approximately half of the topics of one area depend on as many topics from the other) but still important enough to be worth describing. Here the author has tried to draw attention to 'Logic Design' (section 5 — particularly up to logic control) as an area of some importance to three groups: 'Programming Language Structure', 'Programming Techniques' and 'Architecture'. As the reverse is also true, it can be said that these four groups are fairly strongly interrelated (other dependencies within this set have been described earlier). At this stage, the importance of 'Theory' to both programming sections is illustrated.

The last figure (Fig. 3.4) shows how all of the topics relate to each other (a combination of the previous 3 figures). In an attempt to keep the diagram from becoming unreadable, it has been simplified somewhat by allowing some prerequisites

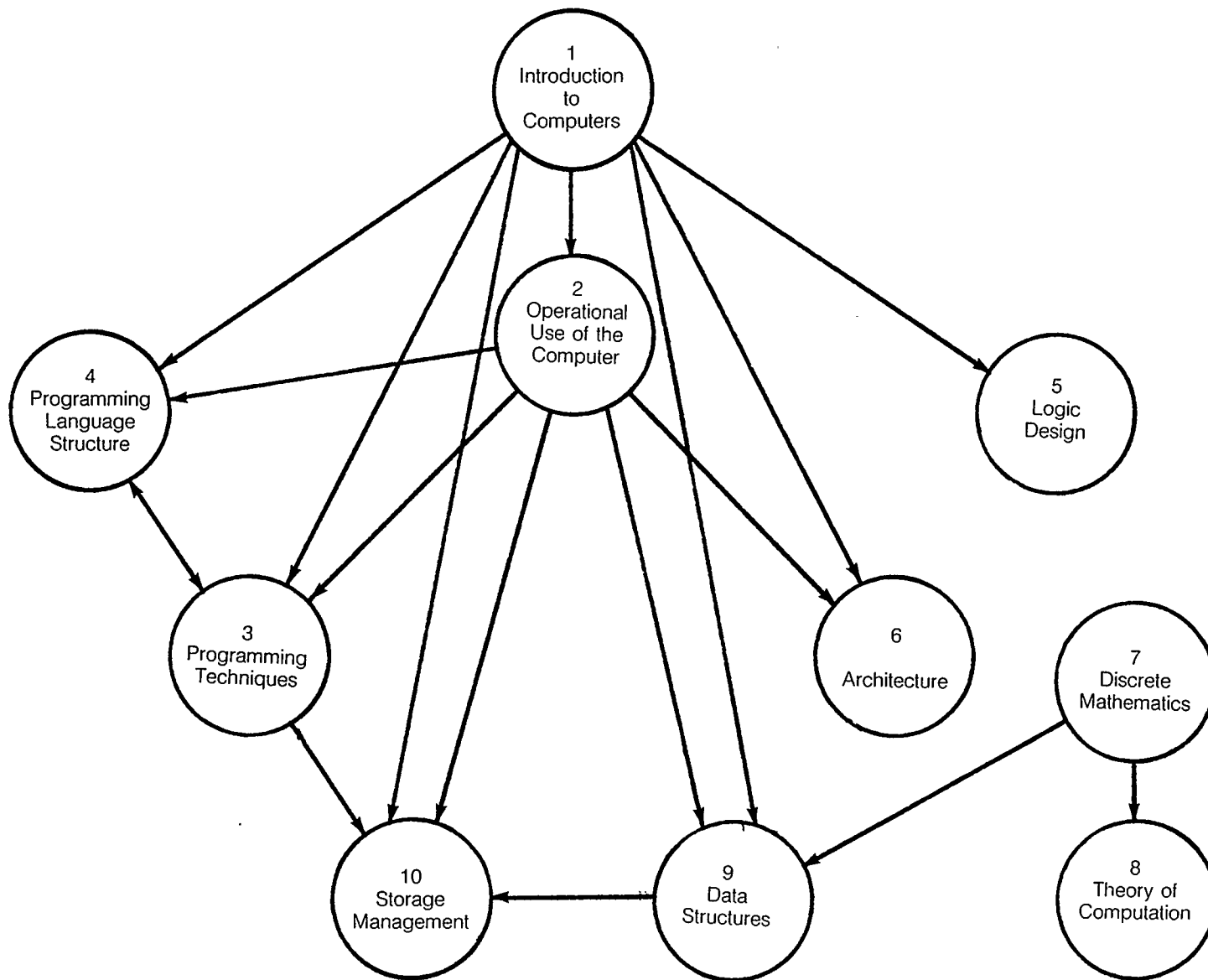


Figure 3.1 Very Strong Interdependence of Topics

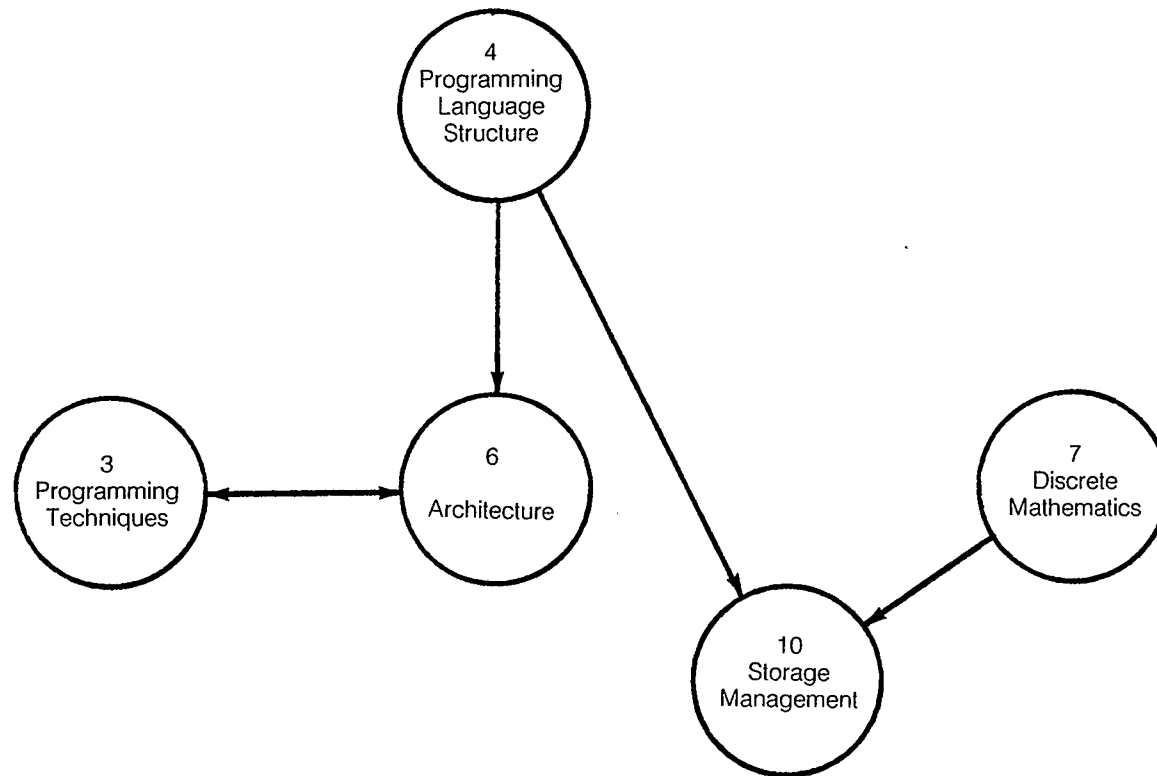


Figure 3.2 Strong Interdependence of Topics

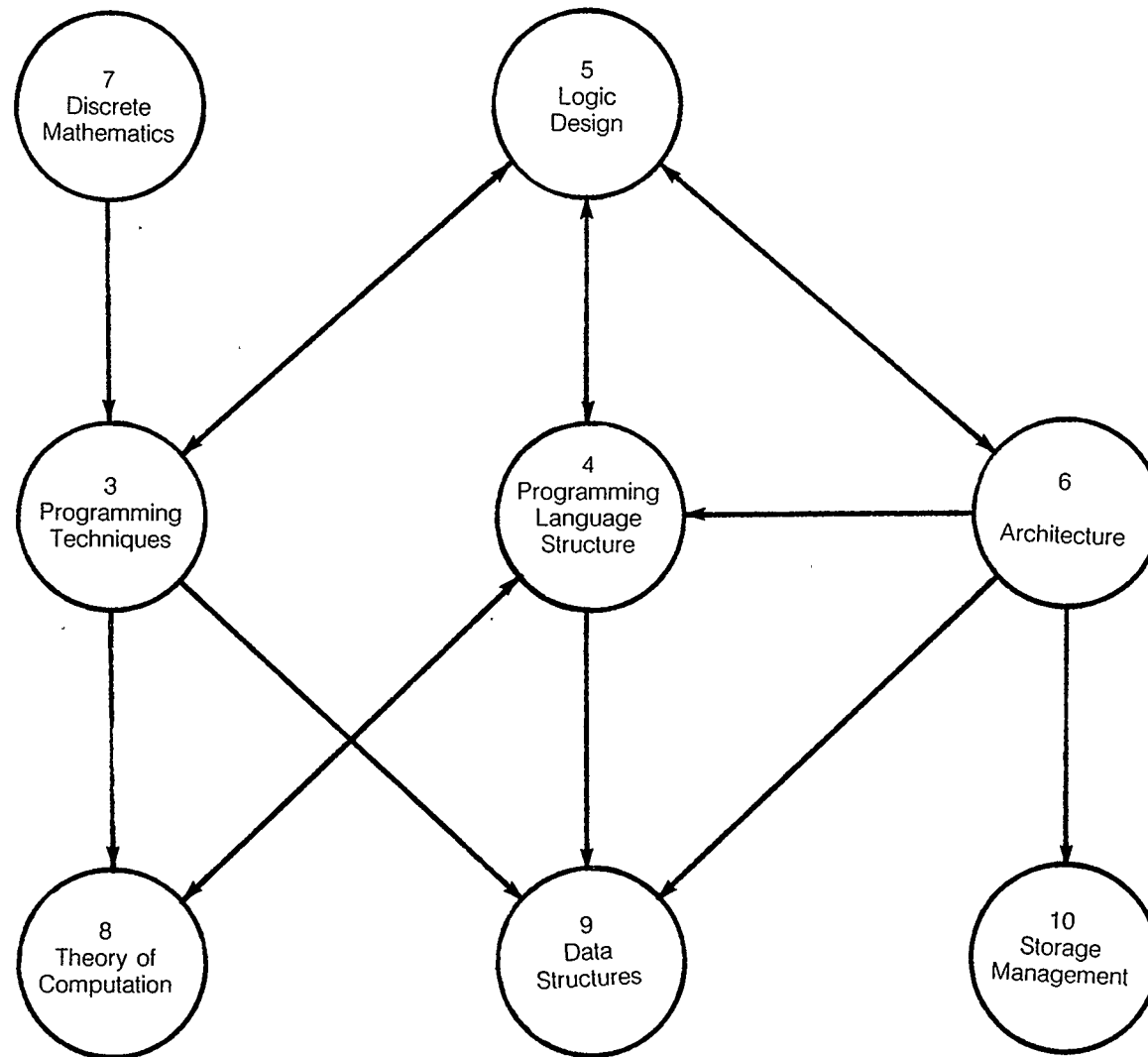


Figure 3.3 Some Interdependence of Topics

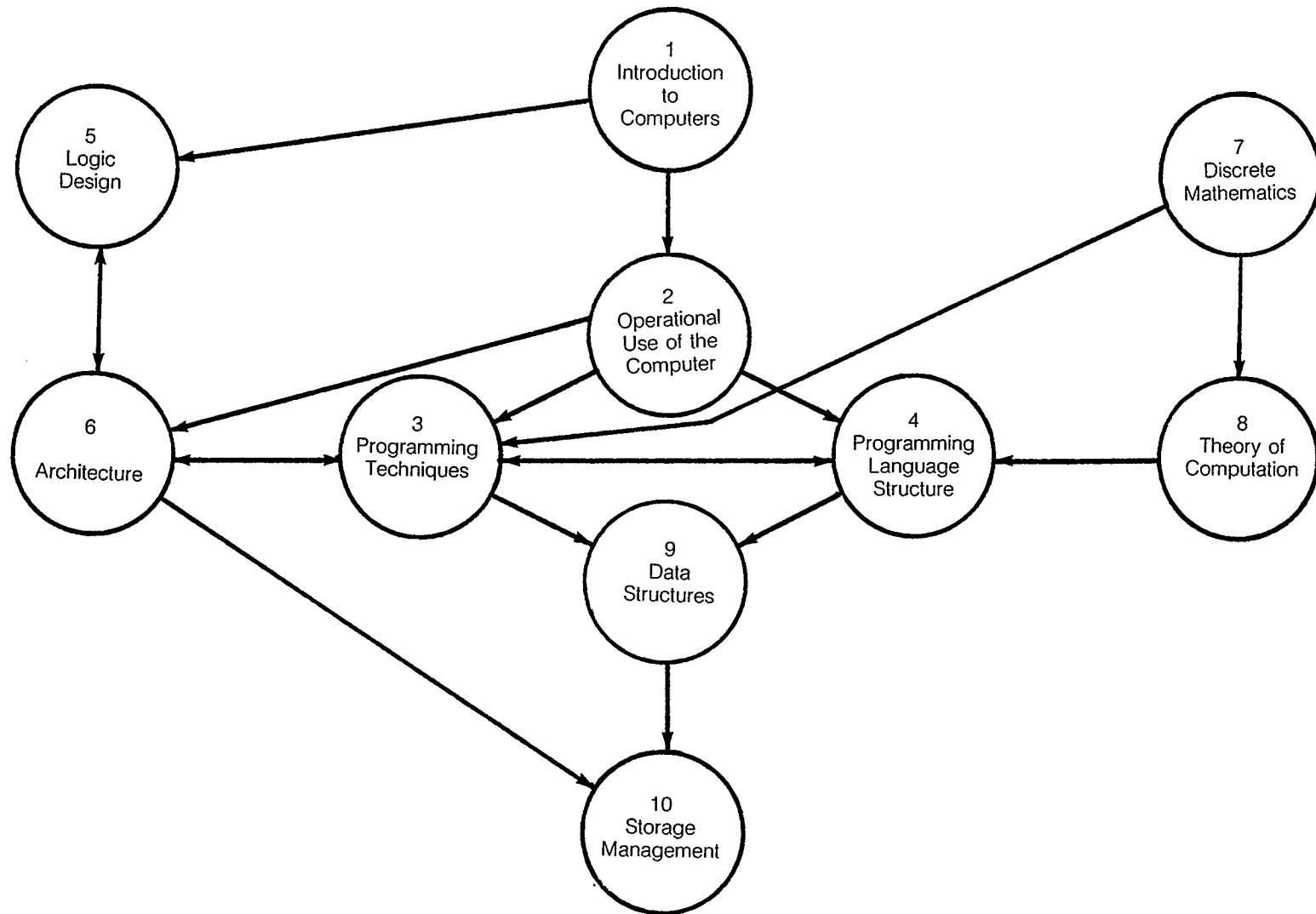


Figure 3.4 Final Structure — Interdependence of Topics

to become implicit. For example, section 2 is a prerequisite for section 9, but also for section 3 which is itself a prerequisite for section 9. It can then be assumed that the connection between 2 and 9 is implicit through 3: there is a transitive relationship.

From these four diagrams, it is possible to draw an outline of the core elementary computer science curriculum. One can not only decide on the order in which the topics should be presented, but also the prerequisites and co-requisites that should be required to provide the fullest possible appreciation of the subject matter. Although this provides a much clearer picture of the elementary curriculum than the list of topics and objectives by itself, it is still by no means complete; it simply outlines the interdependencies of the areas as sections. Each section need not, and indeed should not be covered in its entirety before going on to another one. As an example, the foundations of data structures can in fact be laid before the student has a strong knowledge of programming, so the fact that it occurs second to last in figure 3.4 is somewhat misleading. Indeed, only the more involved aspects need to wait until then.

The areas themselves are not quite as neatly defined as the diagrams would imply either. 'Programming Techniques' for example covers a very broad range of topics, many of which should be discussed throughout the entire curriculum and not just for the middle part of the elementary curriculum.

As stated before, this description is by no means complete, but it is hoped that it does describe the major interdependencies. It leaves enough room for individual preferences, but still, if followed fairly closely, will provide a solid, consistent, and robust web of information that is vital to the education of a well rounded student of computer science.

Chapter 4

QUESTIONNAIRE RESULTS

In order to determine the extent to which existing programs in Western Canada are oriented to convey to their students such concepts as appear in the list (see appendix), faculty members of computer science departments in the universities were interviewed first and then given the previously mentioned questionnaires. Eight universities in all were interviewed, and seven received the questionnaires (one declined). From the interviews, and with the help of the university calendars, it was possible to produce a course outline for an average elementary program. In the following pages, these outlines are discussed and then the results of the questionnaires are covered.

First, a general overview of the computer science programs at the eight universities was drawn up. Using the university calendars, supplemented by the interviews and various other materials (such as course outlines, student guides, etc.), it was possible to draw relatively typical course plans that most students would follow in the elementary curricula. These are shown in Figures 4.1-4.8. The total number of courses usually taken in the first two years is roughly equivalent (when full courses and mathematical courses are counted) in all major, four-year programs (please note: Lethbridge does not have a major's program as yet). The amount of required mathematics varies a great deal; from a very heavy requirement that outweighs the computer science courses in the elementary program at the University of Saskatchewan and the University of British Columbia, to a relatively light requirement at Simon Fraser University and the University of Calgary, where the number of computer science courses the students are expected to take is much higher. The other universities that fall somewhere in between are basically pleased with their situation and although the importance of mathematics to computer science cannot be denied, perhaps, as with almost everything else that is beneficial, these universities feel that moderation is the best approach.

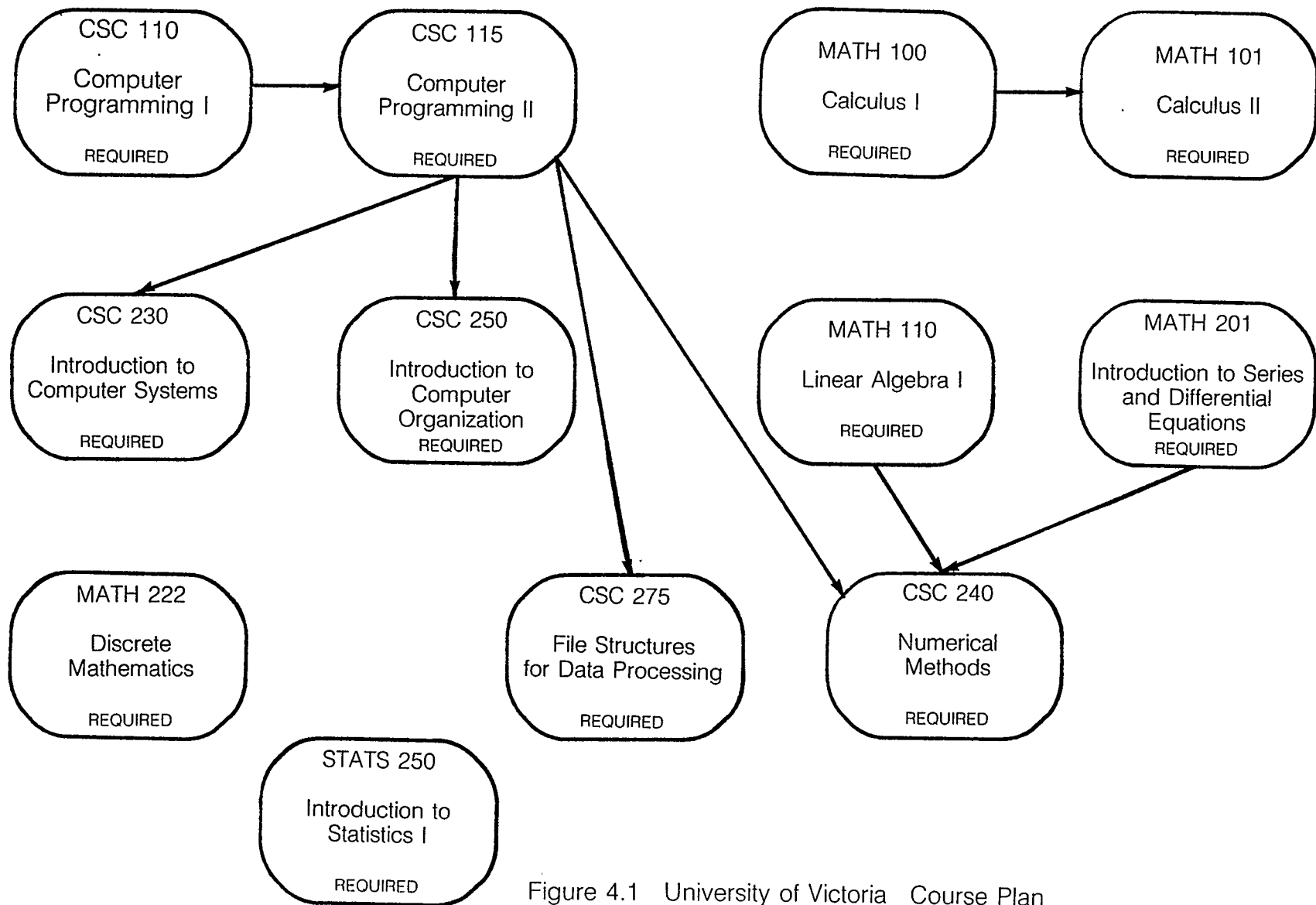


Figure 4.1 University of Victoria Course Plan

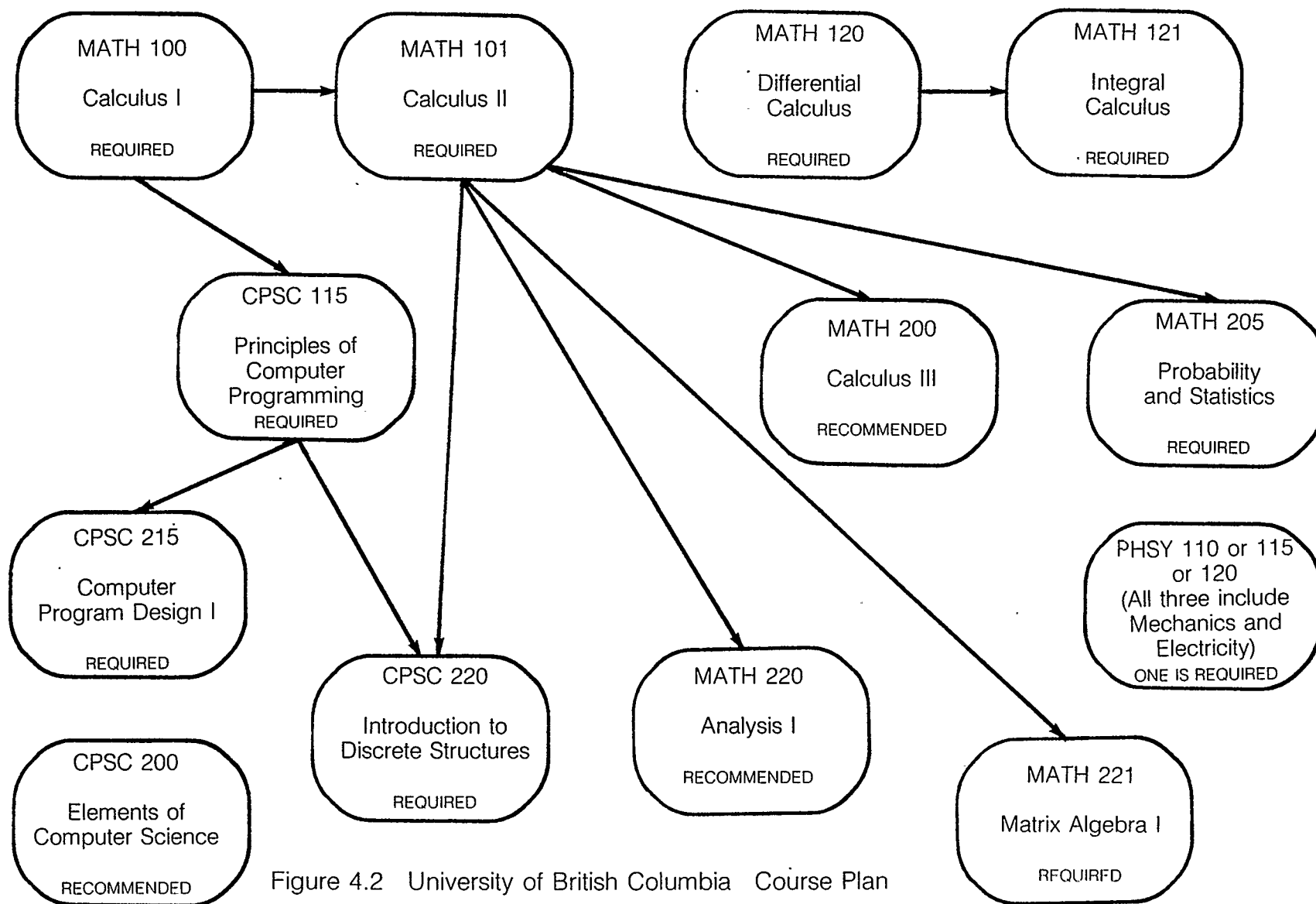


Figure 4.2 University of British Columbia Course Plan

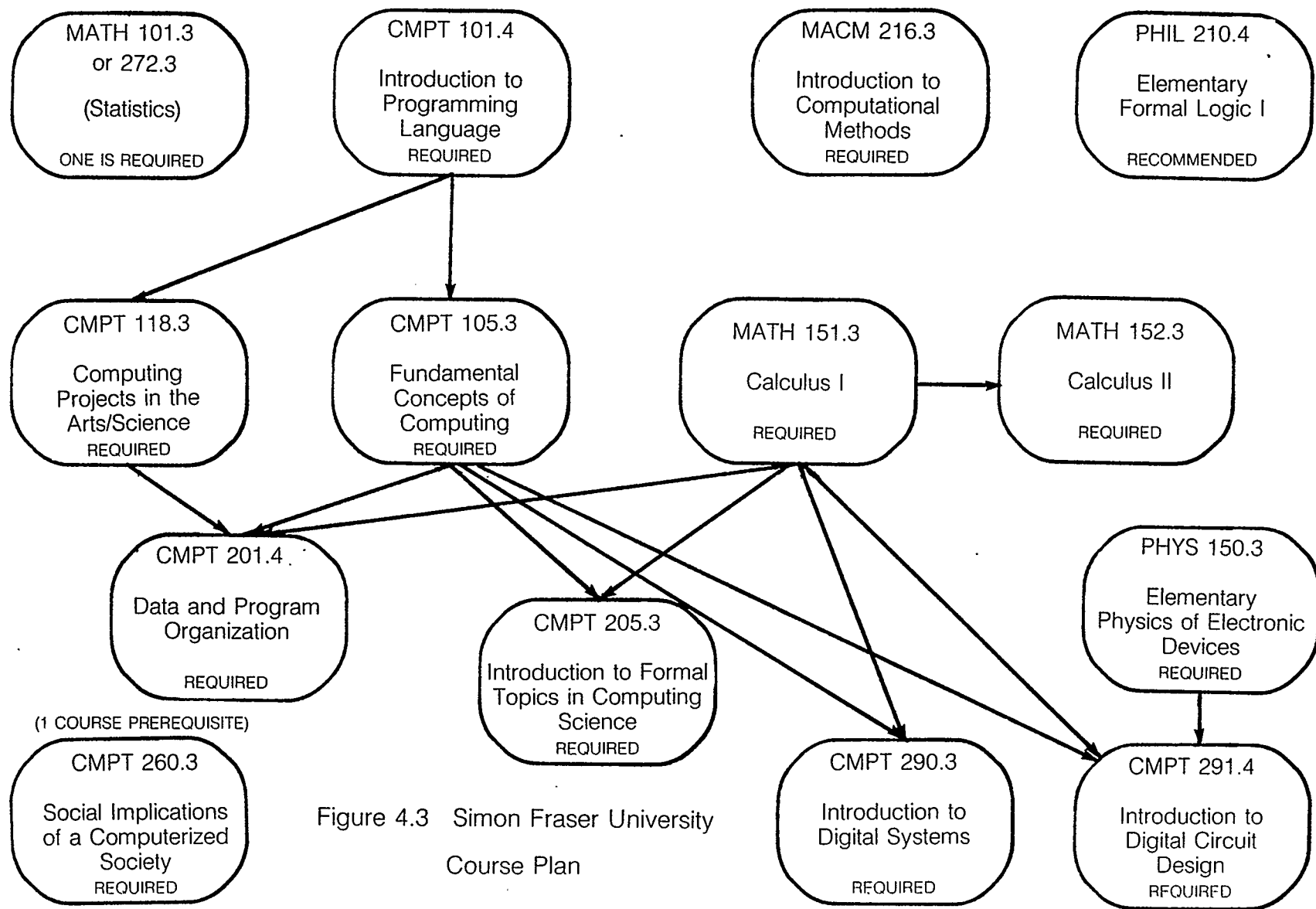


Figure 4.3 Simon Fraser University
Course Plan

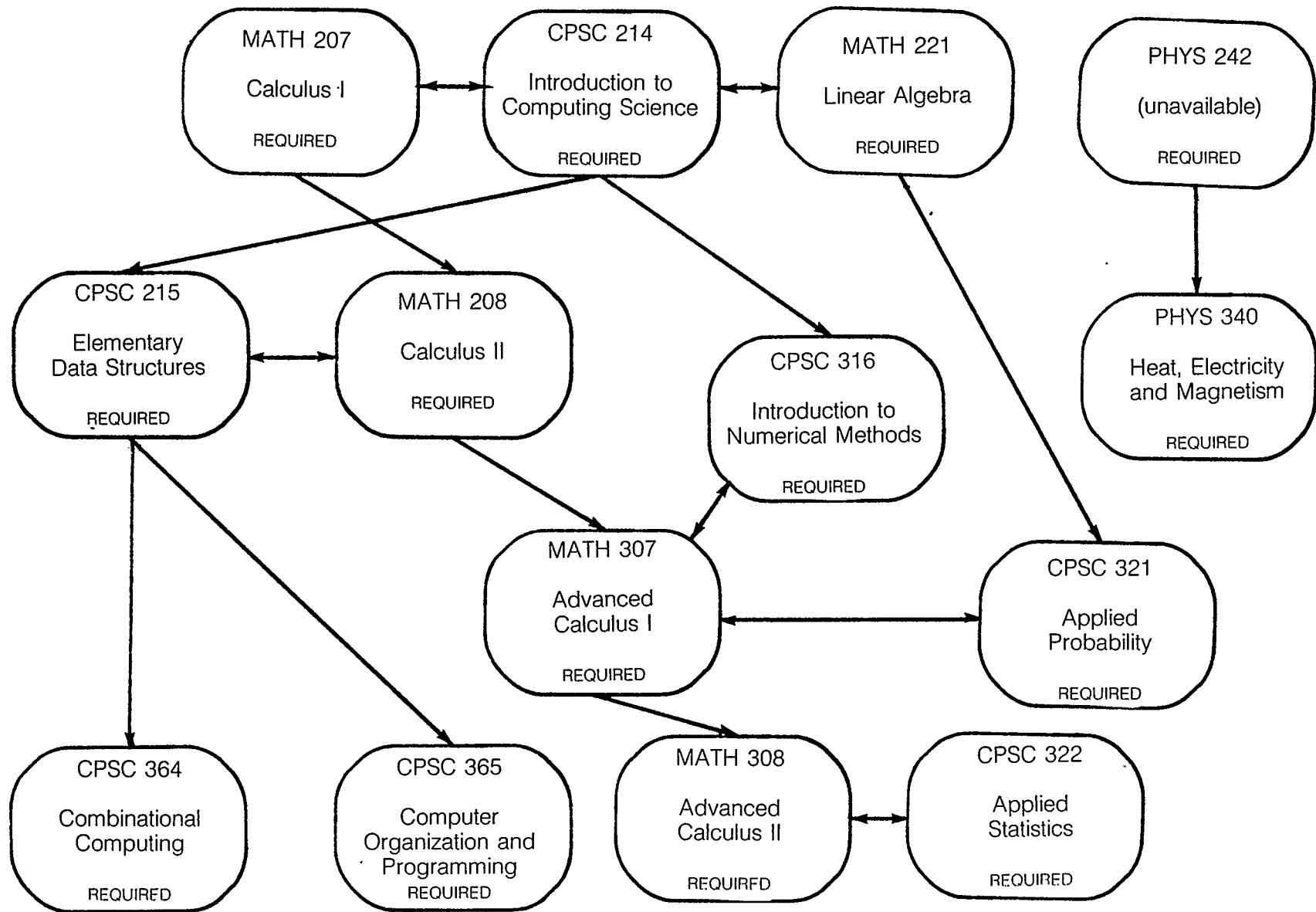


Figure 4.4 University of Alberta Course Plan

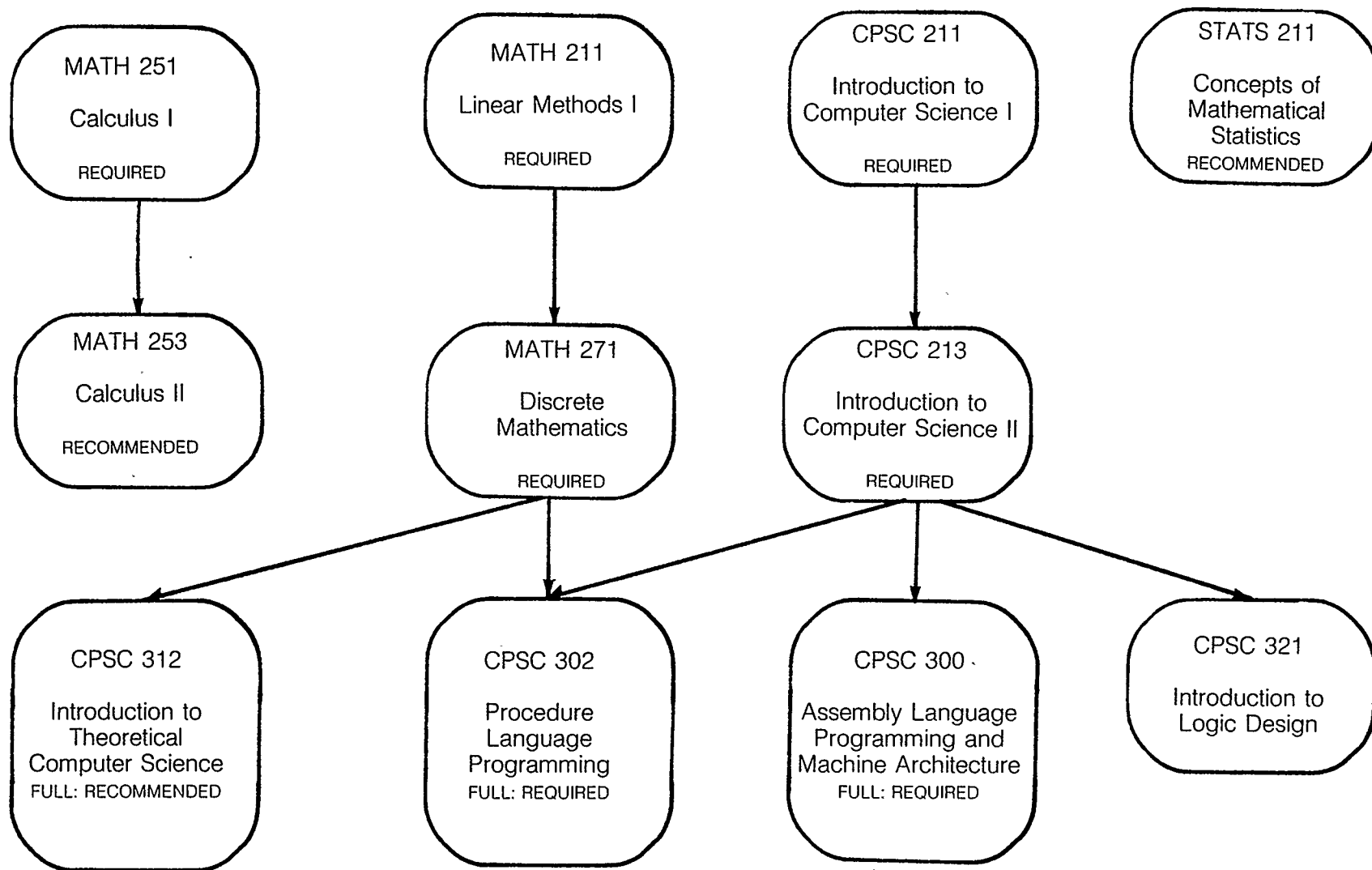


Figure 4.5 University of Calgary Course Plan

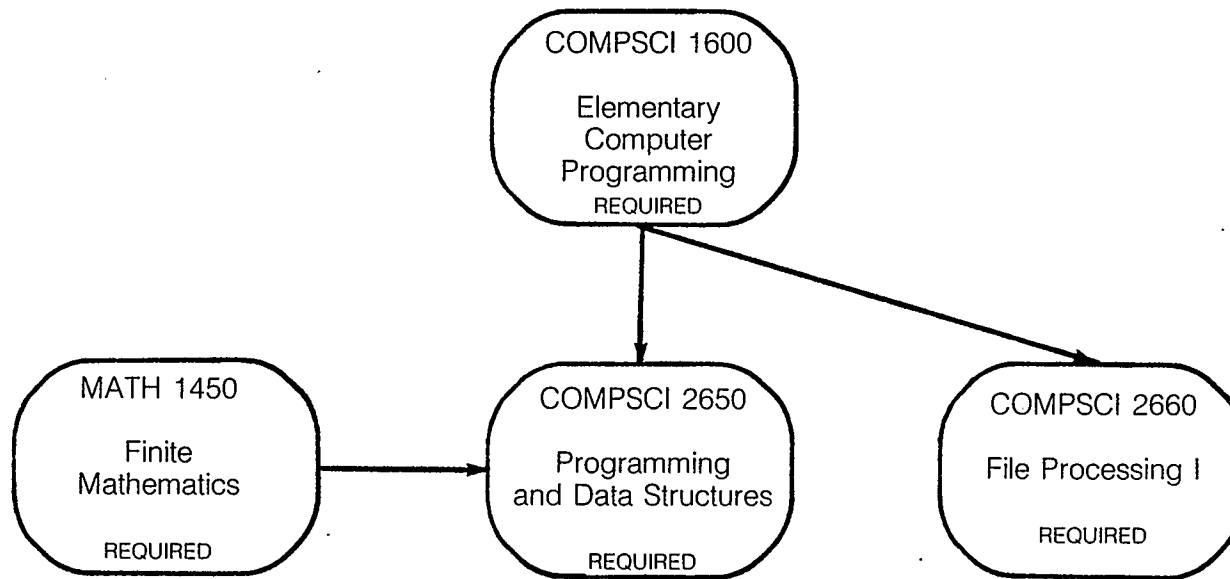


Figure 4.6 University of Lethbridge Course Plan

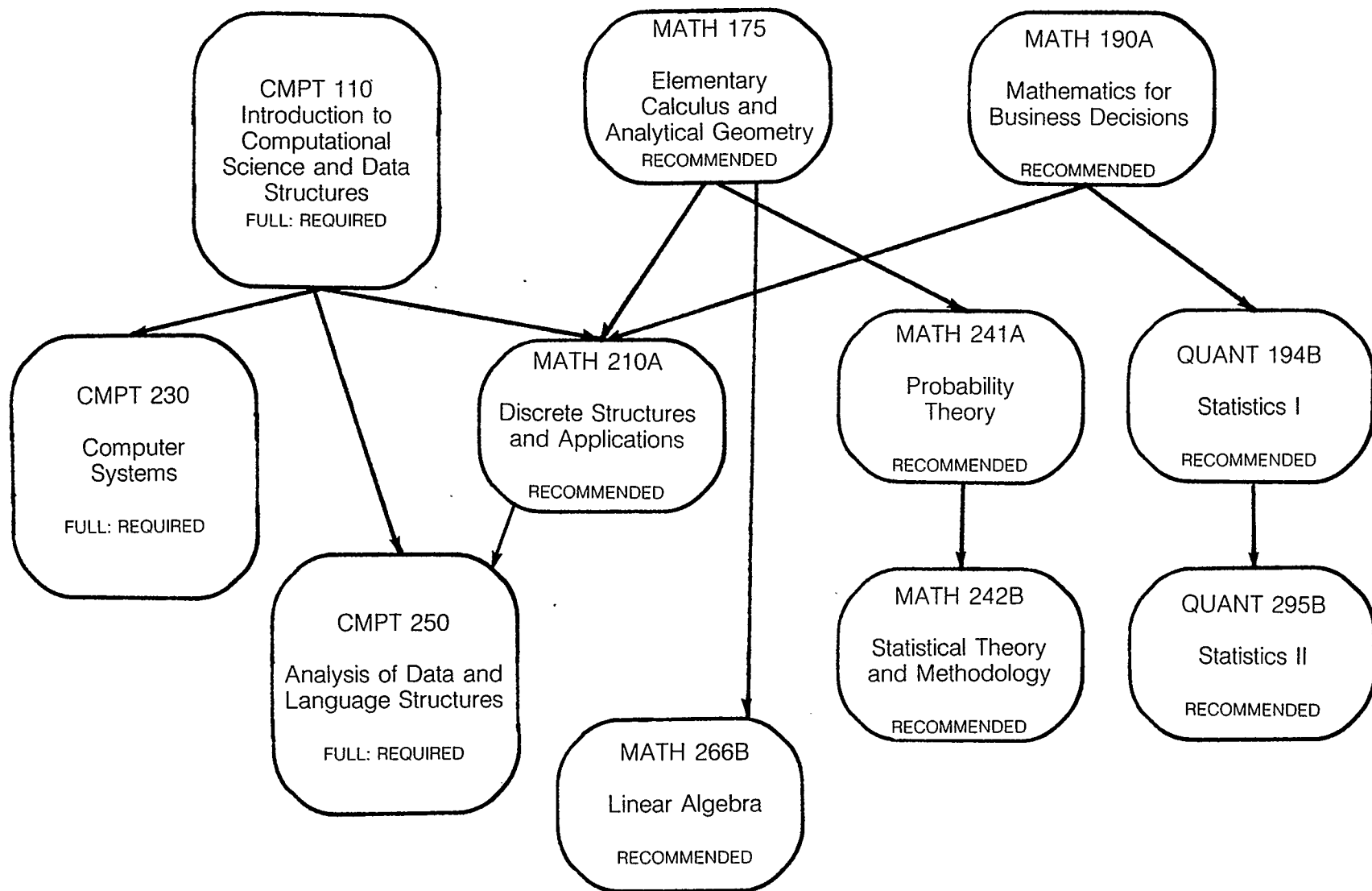


Figure 4.7 University of Saskatchewan Course Plan

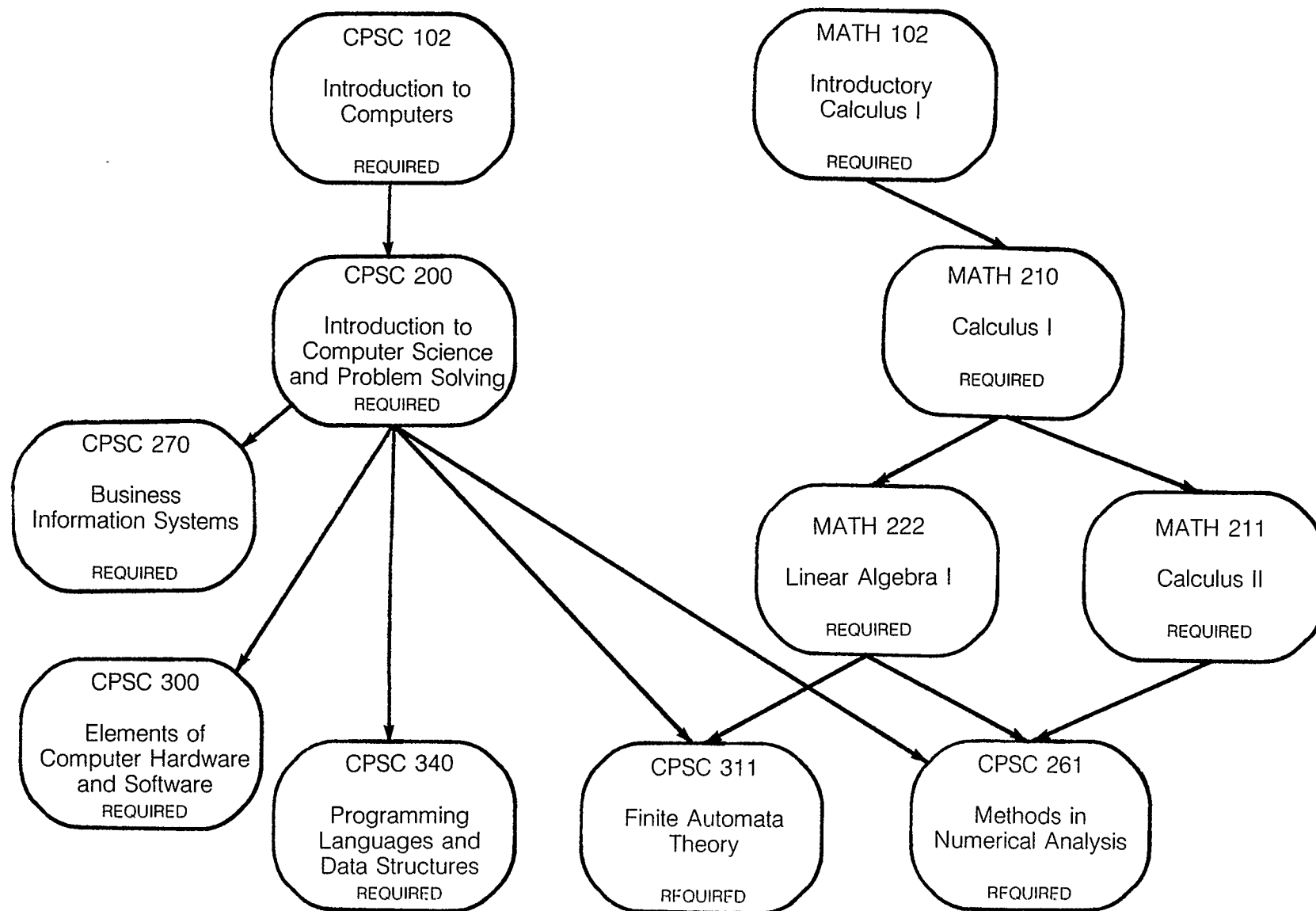


Figure 4.8 University of Regina Course Plan

	UNIVERSITY OF VICTORIA	UNIVERSITY OF BRITISH COLUMBIA	SIMON FRASER UNIVERSITY	UNIVERSITY OF ALBERTA	UNIVERSITY OF CALGARY	UNIVERSITY OF EDMONTON	UNIVERSITY OF SASKATCHEWAN	UNIVERSITY OF MANITOBA
PROGRAMMING	X	X	X	X	X	X	X	X
DATA STRUCTURES	X	X		X	X	X	X	X
ARCHITECTURES	X	X	X	X			X	X
LOGIC DESIGN	X		X		X			
THEORY			X		X			X
NUMERICAL ANALYSIS	X		X	X				X
DISCRETE MATHEMATICS	X	X	X	X	X		X	
LINEAR ALGEBRA	X	X		X	X	X	X	X
CALCULUS	X	X	X	X	X		X	X
STATISTICS	X	X	X	X	X	X	X	
FORMAL LOGIC			X					
BUSINESS APPLICATIONS						X		X
ELECTRICITY AND MAGNETISM		X	X					

Figure 4.9 Topics Covered in Elementary Curriculum

Using university calendars primarily, a summary of the major topics covered by the eight universities in first and second year was created and then this was combined to show what had been covered at the end of the elementary curriculum (Fig. 4.9). Every university covered aspects of programming and either required or recommended at least one course in statistics. With one or two exceptions in each category, most of the other topics were also required or recommended in that same period. Very few (one or two) required their students to study physics (electricity and magnetism), formal logic, or business applications of computer science, while slightly more (three or four) expected their students to study numerical analysis, the theory of computation, or logic design.

When all areas are taken into account, it would seem that the universities with the most complete elementary curriculum are: the Universities of Victoria, Calgary, Regina, and Simon Fraser University. When only the areas specifically from computer science are included, the same four universities fare equally well, each missing only one area. Although this guide may be quite informative, it must still be taken with a grain of salt. At the University of Calgary for instance, some students take a course from philosophy, thus filling the lack in the area of formal logic, but as this is not strongly recommended, it is not included. It is likely that with few exceptions, each university offers courses in all areas at the elementary level, but this is not reflected in the calendar entries for computer science, and so it will not be discussed further. The university calendars were used as the primary source, and it is a well known fact that they often do not accurately reflect on the actual course content, and in addition, when individuals are interviewed, their responses are often tempered by their own personal opinion of a subject's relative importance. This summary then, can be taken as a rough guideline, but by no means is it meant to be the definitive work on the completeness of the elementary programs in computer science departments of Western Canada.

After having listed all of the topics considered by the literature to be part of the elementary curriculum, it remained to discover whether, and to what extent, various universities deal with these topics. This was done by means of a questionnaire, formed from the list of topics and objectives itself. The questionnaire became quite long, but it seemed there was no way to avoid this if there was to be any hope of acquiring a thorough response. The questionnaire was delivered by hand or mailed to the major

universities in the 3 western provinces (7 universities in all) in sufficient numbers so that each full-time faculty member could answer one.

For each topic or objective listed, the instructor was asked to judge the degree of coverage (Fig. 4.10), name the course in which the topics were covered, list the tools and techniques used to bring these ideas across (see list, Table 4.1 — at the end of the chapter), and estimate their satisfaction with the results (Fig. 4.11). From this survey, it was hoped it would be possible to arrive at a list of topics covered and the tools currently used. To supplement this, I also visited the universities to interview various members of the faculties. The results of this survey are described below.

1. not dealt with
2. received superficial mention
3. general discussion
4. in-depth study

Figure 4.10 Degree of Coverage

1. very satisfied
2. generally satisfied
3. sufficient
4. somewhat lacking
5. unhappy with results
6. attempts failed

Figure 4.11 Satisfaction with Results

The questionnaires were given (93 in total) to seven of the eight universities (Simon Fraser did not wish to receive any) and of the seven, five returned completed questionnaires (20 were returned). No responses were received from the University of British Columbia or the University of Saskatchewan.

The above caution about the subjectivity of individuals' responses must also be taken into account when discussing the questionnaire results. The response given for degree of coverage may be tempered by an individual's opinion of its importance. Also, it is understandable that instructors are reluctant to admit when their attempts at dealing with a particular topic were unsuccessful. Unfortunately, there are also some who are insensitive to the students' reactions, and as a result, feel that all they do is

successful. Clearly, none of these problems can easily be deduced from questionnaire results and so because of the subjective nature of the questionnaire and in part because the time and space available for discussion is limited, for the most part, only the most common responses will be discussed. There must however be an exception. Since most people, for whatever reasons, are reluctant to admit failure, those that do deserve special mention. These comments will be dealt with in the next chapter.

Coverage on the first section ('Introduction to Computers') ranged mostly from superficial mention to general discussion with the most common being the latter. These topics were covered primarily in the first year but also to some extent in the second. The tools and techniques most frequently employed were lectures and so-called hand-waving explanations while textbooks and the use of overhead projectors were also found to be present. The term 'hand-waving' is used here to imply a much more animated approach to lectures than verbal explanations and blackboard oriented lectures alone, and is not meant in a derogatory sense. A few included written and programming assignments, and some less common tools were mentioned. The latter will be discussed at length in a later chapter (Chapter 6). By and large, instructors dealing with this area were generally satisfied with the results. There were a few parts where some thought the treatment was somewhat lacking, but again, this will be left to a later chapter (Chapter 5).

'Operational Use of the Computer' was dealt with somewhat more thoroughly than the previous section. While most still classified their coverage as general discussion, some considered they had come closer to an in depth study. By far the majority covered these topics in the first year with very few adding to it in the second. The tools and techniques used were almost the same as in the first section. But no-one listed written assignments as being used. Only the University of Regina dealt with batch use of the computer, and this was done in the second year (the students learn about interactive systems first). A few other tools were used by isolated individuals, but again, discussion of these is deferred to Chapter 6. The degree of satisfaction with the treatment of this topic is evenly split between generally satisfied and sufficient.

When tabulating results in most of the other sections, all topics in a particular section have results that are similar but in the third section ('Programming Techniques'), three distinct groups were formed. All topics in this list from the first (problem solving) to debugging and verification formed one group, all with similar responses on the

questionnaire. The next group included topics from abstract data types to software communication (inclusive), and the rest of the topics formed the third. All topics were covered to some extent in both first and second year. Coverage was primarily general discussion but tended towards a more in depth study for everything but the section from 'abstract data types' to 'software communication', for which coverage was much more superficial. This part was covered using mostly lectures and hand-waving explanations, with a little use of textbooks, overheads, and programming assignments while these were used to a far greater extent with the other topics, including some use of written assignments as well. Satisfaction with everything up to 'software communication' ranged from sufficient to somewhat lacking while satisfaction was generally higher for the rest. Fewer instructors dealt with the topics from 'abstract data types' to 'software communication'.

'Programming Language Structure' was most often left until the second year and the coverage was for the most part on in-depth study (of the objectives stated). Again lectures and hand-waving were the most popular with textbooks and overheads a close second. Some use of both written and programming assignments was recorded and most were generally satisfied with the result.

The next four sections (see Fig. 2.1 for a list of sections) of the questionnaire received a relatively poor response (fewer than 5 responses) which reflects to some extent (particularly with logic design and theory) the results from the summaries discussed earlier.

'Logic Design' was split into two groups, with more answering those parts dealing with data representation, numbers, and digital arithmetic than any of the others. These three were covered pretty well in depth, evenly split between first and second year, and the tools included: lectures, hand-waving, overheads, textbooks and some assignments (mostly written). Most were generally satisfied. The other topics were not covered as often, nor as thoroughly and the satisfaction ranged more from sufficient to somewhat lacking.

The architecture section was next. Most topics were covered by general discussion and some in-depth study of those topics related to assembly and assembler languages. Some coverage occurred in the first year, but more in the second. The most frequently used tools were lectures and textbooks, with some use of hand-waving

explanations, overheads and programming assignments. Those that answered this part of the questionnaire were fairly satisfied with their results.

No-one dealt with matrices and vectors; presumably that is assumed to be covered in linear algebra rather than discrete mathematics. The other parts are covered by general discussion, in the second year, using lectures, textbooks, and both written and programming assignments. It was generally thought that the results were sufficient.

There was very poor response (few people answered) on the theory section but those that answered had the following responses. Finite automata, formal languages, context-free, and context-sensitive grammars are dealt with superficially; turing machines and computability are studied in a fair amount of detail, and analysis of algorithms received general discussion. Information theory is not covered in the same courses as the rest of the topics. The others were dealt with in the second year only. They were taught using lectures, textbooks, and written assignments with a sufficient degree of satisfaction.

The section on data structures had a much better response. The topics in this category were covered by general discussion and in-depth study in the second year using lectures, textbooks, and programming assignments. For the most part, the degree of satisfaction was sufficient. The same can be said of the next section on storage management although the degree of satisfaction is somewhat less.

Although some listed all parts of the major topics in section 11 as having received at least superficial mention, only the first four (numerical methods, word processing, data processing, and random numbers) were covered by more than four individuals. The topics were quite evenly spread between first and second year and all of the traditional methods (lecture, hand-waving, textbook, overhead, and assignments) were used as tools with varying degrees of satisfaction (from generally satisfied to unhappy with results).

In the first section, 'computers and the law' together with 'the computer industry' were hardly discussed at all and a number of individuals expressed their dissatisfaction with this situation. The sections on history and social issues did receive more attention both in the forms of superficial mention and general discussion in both years using mostly lectures and hand-waving with an average degree of satisfaction.

In summary then, most topics were treated to a general discussion in both years

with the main emphasis in the second. Students get most of their information through lectures, textbooks and blackboard demonstrations. It would seem that they are seldom given an opportunity to actually work with the information they have been given with the exception of some assignments. The respondents had a choice of 20 tools and techniques on the questionnaire (see Table 4.1), but with few exceptions (discussed in Chapter 6), only six of them (1-4, 9, 10) were used; the specifically designed tools seemed almost totally left out. Many educators agree that the value of doing versus seeing and hearing is not to be underestimated [Good80]. Although the instructors were for the most part satisfied with their efforts, there were some areas where more than one instructor was not. As stated earlier, these responses are quite important and will be dealt with in the next chapter.

Table 4.1
TOOLS / TECHNIQUES

1. blackboard-oriented lecture — fairly self-explanatory — includes verbal explanations and notes, diagrams, etc. written on the blackboard
2. hand-waving explanations — implies a somewhat more animated approach to explanations that can sometimes be very effective
3. textbooks — also covers use of supplementary reading material
4. overhead projectors — allows use of previously prepared slides, copies of which can be given to the students in the form of handouts
5. guest speakers — includes other faculty members, professionals from industry and others — most effective when speaker talks about a subject that can be considered their own area of expertise
6. films — fairly self-explanatory
7. seminar-type discussions — ideally, the discussion will be lead by someone who knows the material well
8. student research and presentations — difficult to accomplish in large classrooms, but manageable in a lab or tutorial setting — important for the students to learn how to research as well as how to speak in front of a group
9. written assignments — hand-written — includes everything from arithmetic problems and short answer to essay-type explanations
10. programming assignments — anything that involves writing and/or running programs on a computer
11. assignments that require the student to demonstrate ability in some activity or skill — eg. use of a particular subsystem

Table 4.1 (continued)

12. assignments that require the student to build something (device) — eg. building a half adder from flip-flops
13. demonstrations using hardware adapted to illustrate the topic — hands-on experience is almost always preferable but one could use demonstrations when the device does not exist in sufficient quantity for the students to use
14. demonstrations using hardware designed to illustrate the topic — systems can be built that illustrate the Von Neumann machine — a large one can be used to show a whole class
15. hands-on experience with hardware adapted for that purpose — eg. older shift calculators can be used to illustrate multiply and divide algorithms
16. hands-on experience with hardware designed for that purpose — eg. DEC Logic Labs (see chapter 6)
17. demonstrations using software adapted to illustrate the topic — software designed for one purpose can be used to illustrate other ideas — eg. simulations can be used to show how simulations work
18. demonstrations using software designed to illustrate the topic — the student can run programs already written that can show how various algorithms work
19. hands-on experience with software adapted for that purpose — eg. on-line documentation systems can be used to help the student familiarize themselves with the computer system
20. hands-on experience with software designed for that purpose — such as LEARN on UNIX [Kern79]

Chapter 5

RECOMMENDATIONS

Using the results of the questionnaire, it was possible to arrive at some conclusions about the areas where supplementary tools would be most beneficial. These areas are discussed and general suggestions are made for the type of tool that would be suitable.

Instructors were basically satisfied with most areas covered in the questionnaire. There were however, a number of topics that were felt to be somewhat lacking and even some with which a few individuals were unhappy. These are the areas with which it is felt that new tools should be developed. With very few exceptions, the same traditional tools are being used to teach all aspects of computer science (lectures, textbooks, etc.), even though the nature of one topic may be drastically different from that of another. The teaching of skills for example, can be approached differently than the presentation of facts or the understanding of concepts [Good80]. If a particular topic can be identified as to its nature (concept, knowledge, skill, etc.), this information can be used in deciding what types of tools would be most appropriate. Many topics are not easily slotted, as they are composed of many different parts, so instructors must be willing to try different approaches. They must also be sensitive to the reactions of their students: simply because a film was successful in illustrating sorting algorithms [CSRG81] doesn't mean that films are always useful. Similarly, lectures may be completely adequate for some topics, while resulting in almost total confusion when used to cover others. For those topics that can be identified, in particular those classified as skills or facts, it is known which methods are most effective [Skem79], and thus development of tools for these should be quite straight-forward.

The following is a discussion of those topics with which instructors were unhappy. Where possible, the topics have been classified and suggestions are made as to which type of tools may be most useful. For a summary list of the areas of need, see Table 5.1.

The initial introduction to computers is typically quite straight-forward but there are always a great many new ideas that the students must learn all at once. In addition to this, for most students, their first computer science course comes in their first semester, and, as a result there are a number of practical problems to be overcome (students must be motivated to think and work on their own — something they often do not learn in high school). There were a few parts where instructors felt the student's comprehension was insufficient. Their understanding of the operation of memory and being able to distinguish functions performed by the different parts of the computer system (O/S, hardware, their own programs, etc.) were felt to be somewhat lacking. In this case, as the topics are largely factual at this level, simply allowing more exposure, perhaps in the form of exercises, demonstrations, and/or models should make assimilation of this information more complete for most students. More exposure (in the form of more time) is not always possible, and many students will tell you that they have little enough time to complete all their work as it is, without adding to it. With carefully designed exercises that guide the students through the work, it is possible to make somewhat more efficient use of the available time than is often done. This is especially true in classes that have scheduled labs or tutorial time that is currently being used solely to answer questions.

There were a few areas under the heading of 'operational use' where instructors felt there was a lack and even a few areas where they felt unhappy with their results. These areas were the students' ability to prepare and run programs and their understanding of the logical subsystems of a computer. The first could be helped by a document that details the process and requirements for preparing assignments and the second by exercises and tutorials that require the students to use the various subsystems and answer questions about them. Again, it is felt that more exposure (perhaps through more effective use of the available time) would produce a noticeable improvement in the skills of many students. If the students are to be expected to recognize the different subsystems, they must be given the opportunity to practice using them and moving between them in a guided fashion.

The other areas in this section where instructors felt the students were lacking were in their understanding of system organization (related to subsystems, etc.) and several important, though often neglected skills. Keyboard skills and terminal operation are skills the students are expected to pick up on their own. It is strongly felt that even a

few hours of guided practice would go a long way towards improving these skills.

Not surprisingly, the main areas of difficulty in 'Programming Techniques' were those that dealt with problem solving, evaluation of algorithms, and programming style — all topics that are difficult to define. Problem solving is a topic that most would agree is difficult to teach as well as master. Indeed, it is more of an art than a skill, and cannot be approached head-on. One can show the student various methods of attack [Poly57] and give them many examples (graduated examples, and sequences of assignments where each one builds on the previous one work relatively well) in the hopes they can build from this a 'database' of problem types and approaches to solutions but it is clear from the students' responses that this is by no means the ideal solution. Perhaps the difficulty that the students experience is due in part (aside from its inherent difficulty) to the fact that the learning environment is neither imaginative nor varied (both important qualities of good problem solving skills). Educators in post-secondary institutions too often assume that their students are all mature individuals, motivated by a burning desire to learn, or that since they themselves find a subject intrinsically fascinating, their students must too, and thus make little effort to provide a stimulating environment so important to all forms of learning.

The other areas (algorithms and programming style), all related to problem solving to some extent, again can be helped by numerous examples. Problems in debugging and verification often stem from a lack of understanding of how programs operate. Perhaps a better treatment of how programs are translated and executed at an earlier stage in the students' careers is one way to improve the situation. One method of bringing this about is through exercises in hand execution of programs (another area where students are lacking). Another is through exercises with syntax [Beck83]. Two such exercises are described in Chapter 7.

Recursion is also a topic with which instructors are often unhappy. Recursion is a difficult concept and most traditional means of introducing the idea fail. Even when the students appear to have mastered specific applications the general concept still escapes them. Examples of recursion that the students are already familiar with (and therefore can easily relate to) are difficult to find. Most often, examples are taken from the realm of mathematics, an area most students find uncomfortable in the first place.

Table 5.1 Areas of Need

Area 1:	Introduction To Computers Components of the Systems The Operating System
Area 2:	Operational Use of the Computer Keyboard Skills System Organization Interactive Use Running Programs
Area 3:	Programming Techniques Problem Solving Algorithms Programming Style Debugging and Verification Manipulating Character Data Recursion
Area 4:	Programming Language Structure Flow of Control Translators and Compilers Run-Time Environment
Area 5:	Logic Design Digital Circuits Digital Arithmetic
Area 6:	Architecture Hardware Systems Organization Memory Symbolic Coding Assembler Language Program Segmentation and Linkage Macros
Area 9:	Data Structures Lists
Area 10:	Storage Management Files
Area 11:	Other Major Topics — (all parts)
Area 12:	Miscellaneous Social Issues in Computer Science The Computer Industry

Several ways of dealing with this subject are discussed in Chapter 6. The programming techniques part is perhaps the single most difficult area to deal with because the skills and concepts being taught are so intangible, yet they are fundamental to success in computer science.

'Programming Language Structure' is much more straight-forward than the previous area, and, as expected, fewer problems were expressed. The main concerns were with 'flow of control', 'translators and compilers' and 'the run-time environment'. These areas primarily deal with factual information and as a result, simulation (by hand or otherwise) of the various processes should yield positive results. As with any other tool, such exercises and systems are time-consuming to develop, and most computer science instructors have little time for such work, so even when a solution is clear, it is often not implemented. Sometimes other systems are adapted for this new purpose, but they are often of minimal use because they are so cluttered with other parts that students miss the point. It is very important to be able to isolate those ideas the student is expected to gain from a tool as much as possible. The intent should be to leave the student to concentrate on the problem at hand, and then integrate the idea in stages into a more realistic environment.

As fewer people answered the next four sections, it was more difficult to come to any conclusions. Those teaching discrete mathematics and theory of computation reported that they had experienced few difficulties. This may be somewhat misleading. Students and instructors alike expect these topics to be difficult and perhaps because of this, a rating of sufficient has a slightly different meaning than for other categories. Practically no tools exist for these topics except the traditional lectures and textbooks. It is quite likely that in this area, traditional tools work very well, however, it is still felt that development of a few tools for demonstrations and practice purposes could be very enlightening.

In 'Logic Design', it was felt that the treatment of digital circuits was somewhat lacking as well as parts of digital arithmetic — in particular: multiplication, division and floating point. These topics can be considered factual, and as such, demonstrations and exercises can be of benefit. Several tools are currently available to show (from a computer science point of view) how these circuits operate and they will be discussed in Chapter 6. Multiply and divide algorithms can also be demonstrated in various ways. These too will be described in the next chapter. Floating points representation and arithmetic could benefit from some tools developed for demonstration purposes as none currently appear to exist.

In the area of architecture, although most were satisfied, some expressed

dissatisfaction with some parts. Overall hardware systems organization and bootstrapping presented some problems. These involve the understanding of certain concepts and thus are not as easily tackled as facts or skills. Perhaps demonstrations and hands-on experience with real or simulated systems (animation might also be used to advantage) would be of service. Another area found to be lacking was memory (access, control, paging, and segmentation). Again, as this information is largely factual, demonstrations should be useful. So much of computer science involves knowledge or understanding of dynamic processes yet most of the tools used are of a decidedly static nature (blackboard or overhead drawings, books, computer program listings, etc.). On closer inspection, these tools are also found to be passive, with little for the student to do but listen and watch. The value of doing versus seeing and hearing must not be underestimated.

Instructors were found to be displeased with their results with the students' understanding of 'pseudo-ops'. This is suspected to be due to insufficient time available to spend on the subject rather than approach. Assembler language was another subject area where some were unhappy. Problems in this area are probably related to difficulties encountered in section 3 (programming techniques) and as such the solution should be related to those for problems in Section 3. Several questioned the success of their attempts to teach loading and linking as well as relocation. Perhaps this is also due to the dynamic nature of the process and the static state of the available tools. Macros almost invariably give students trouble, and a few instructors mentioned this. Some of the problem may again be related to section 3 (with similar solutions), but the rest could be aided with numerous examples (perhaps graduated examples). Macros are a rather difficult concept to master as well as teach. It is somewhat disheartening, with so much discussion going on about the importance of well structured languages (such as ADA, for example), that most macro languages still have syntax that is so awkward and cumbersome that few learn to use them effectively.

'Lists' was the subject that caused the most trouble in the section on data structures. Topics in data structures do not easily fall under the categories of fact, skill, or concept: all of these play their role. One method for dealing with them is suggested in Chapter 8.

In storage management, the section on files was found to be somewhat lacking. Much of what is covered at the elementary level is factual, and this type of information can be covered quite adequately using demonstrations and exercises.

In each area covered under major topics, at least one respondent stated they were dissatisfied with the results. The main reason for this was lack of time for coverage so the students did not have a chance to assimilate the information given. The same comment was made for the last section ('Miscellaneous Topics') with the results of their dealings with the social aspects and the computer industry. Several films on these subjects exist which are quite well made ('Fast Forward' series), and it is felt that often the primary source of difficulty in these areas is lack of organization on the part of the instructor (for which the instructor is not necessarily to blame). Most instructors have very little time to spare, and these topics are not usually considered to be part of the core material. As a result, these topics do not receive as much attention as perhaps they should. The remedy for this situation would be either to provide a finished package that the instructor could use, or to create (somehow) more time for the instructor.

With each topic considered in this chapter, it becomes apparent that many problems are related to each other and a solution for one will go a long way towards a solution for several others. The dynamic nature of many of the processes deserves special note, and it is felt that recognition of the importance of this fact will lead the way for development of tools that provide the stimulation necessary to overcome the difficulties. Hands-on experience is a vital asset and with proper guidance, it can fill in many of the holes that still exist. Although very time-consuming, organization of the materials presented is also important, and many subjects could benefit from lessons learned in education [Good80].

Chapter 6

CURRENT TOOLS

Most instructors used only the traditional tools (lectures, textbooks, hand-waving) to bring their ideas across, but a few tried other methods. Those other methods along with other tools currently available are the subject of this chapter.

The results of the survey would indicate that very few instructors employ any but the most obvious tools and techniques. In the institutions studied, almost everyone listed the first four tools (see Table 4.1) as tools they used in teaching and some also included programming assignments and to a lesser extent, written assignments. Very few however (never more than two), listed any other tools or techniques from the list of twenty that were available. The distribution of tools used according to major area is shown in figure 6.1. Although the tools used seem to be spread fairly evenly, it must be noted that in most areas the tools were used for only a few topics and then only by one or two individuals. There are several reasons for this deficiency, lack of time and motivation not the least among them.

One point that deserves mention is that the instructor's satisfaction with any particular topic tends to increase when he uses more and varied tools and techniques. The reasons behind this are not quite clear but it would seem to be a combination of positive feedback from the students and a result of the effort expended.

No tools other than the traditional ones were used on sections 7 ('Discrete Mathematics') and 8 ('Theory'). There was a very poor response on these (few people answered). Any individual instructor is unlikely to have the time to develop many different tools, and indeed, few tools appear to exist in this area. Another possibility is the fact that discrete mathematics and theory have much in common with the other branches of mathematics and, as such, traditional methods are considered to be sufficient.

Some tools were not used at all — among them, student research and presentations (#8, see Table 4.1). No one said they made use of guest speakers. Although it is likely

TOOLS AND TECHNIQUES

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1 INTRODUCTION TO COMPUTERS	X	X	X	X	X	X			X	X	X	X	X		X		X			
2 OPERATIONAL USE OF THE COMPUTER	X	X	X	X		X	X	X	X	X				X			X	X		
3 PROGRAMMING TECHNIQUES	X	X	X	X			X		X	X	X							X	X	
4 PROGRAMMING LANGUAGE STRUCTURE	X	X	X	X					X	X	X			X					X	X
5 LOGIC DESIGN	X	X	X	X			X		X	X					X	X				
6 ARCHITECTURE	X	X	X	X					X	X	X						X	X		X
7 DISCRETE MATHEMATICS	X		X						X	X										
8 THEORY OF COMPUTATION	X		X						X											
9 DATA STRUCTURES	X	X	X	X		X				X	X						X	X		
10 STORAGE MANAGEMENT	X	X	X	X						X	X									
11 OTHER MAJOR TOPICS	X	X	X	X	X	X			X	X	X		X				X			
12 MISCELLANEOUS	X	X		X		X				X										

Figure 6.1 Distribution of Tools Used

to make little difference in some areas, there are a few topics where the use of someone whose area of expertise lies in that field could not only make the topic more interesting, but also more complete (eg. history, or social issues).

Most respondents did not comment on the tools they used even when they were out of the ordinary. Those that were mentioned are discussed below.

Films were used by some for the introductory material, lists ('L6'), sorting algorithms ('Sorting Out Sorting'), other major topics, and computers in industry. Some material changes fairly quickly, making films as well as other materials out-dated too fast to make their production worth-while, however films, especially animated films provide an ideal medium for illustrating various algorithms (such as sorting) and other complex concepts. The dynamic nature of some of these topics can probably be illustrated best by the use of animation.

Seminar type discussions were used for parts of sections 5 ('Logic Design') and 11 ('Other Major Topics'). Due to the large numbers of students in most universities, it is not surprising that this technique is not used more often. However, at those universities that have small labs (with 25 or fewer students), it would be possible to conduct seminars and even presentations in the labs (provided the teaching assistants are reliable).

Assignments that require the student to demonstrate ability in some activity or skill (eg. use of a particular subsystem) were used quite frequently by one individual and occasionally by one or two others. By their very nature, most programming assignments also require the students to demonstrate skills and abilities, but few recognize this fact and take advantage of it. In order to complete a programming assignment, students are usually required to run the programs on the computer. Being able to enter and run programs involves skill in the use of the system. By adding a few questions to an assignment that directly involve using the computer one can expand and build on these skills.

The rest of the tools fall into two main categories: hardware and software tools. Neither will usually be composed of just one or the other (many hardware tools run various kinds of software, and of course, all software must run on a machine), but these tools are categorized according to their major components. As can be seen in the figure (6.1), various tools were used in almost all areas but usually by only one

respondent and most of them did not explain what those tools were. Among the tools named were a CAI system (a software tool) called LEARN, which runs on UNIX [Kern79], used in some introductory courses. One instructor at the University of Calgary recommended to his introductory students that they use an on-line documentation system called BROWSE [Bram83] in order to familiarize themselves with the system (UNIX). BROWSE is a highly interactive, friendly system which helps introductory students gain confidence in themselves and their ability to help themselves.

Other software tools used by at least one instructor were programs that served as examples to run, sample code for students to examine and simulations to study. Most instructors teaching any of the software topics use examples, and they are almost always appreciated by the students, but to produce three or four running programs for each new topic requires a great deal of time and effort, and as such, must be considered a very important tool indeed.

Although not mentioned by those who responded to the questionnaires, many other software tools exist, and some of these will be described in the second half of this chapter.

Very few hardware tools were named by the respondents. The logic labs from Digital Equipment Corporation were one set used for some aspects of logic design. They are described later in the chapter. Older shift calculators were used by several instructors to aid in explaining multiply and divide algorithms. Although shift calculators are not really in the category of hardware, they are still close enough to fall in that group.

One other tool was used by one instructor which can not be classed as either hardware or software. Wooden models of the 'Towers of Hanoi' were used by one instructor in a second year programming course to help the students grasp the concept of recursion.

In addition to other software tools that are available but not mentioned, there are also numerous hardware and other tools, some of which are described in the following pages.

6.1 Software Tools

Software tools includes all tools whose main component is a program or series of programs. The hardware that this software runs on is, of course also important, but in many cases, the same system can be run on several different machines.

SP/k

SP/k [Holt77] is a programming language designed specifically for teaching purposes. It is easy to learn, yet it makes the transition from a teaching language to a 'real' language quite simple: SP/k is a compatible subset of PL/1. Designed to encourage structured problem solving by computer, SP/k is actually a sequence of eight subsets. Each subset introduces new constructs (see fig. 6.2), while retaining all of the constructs from previous subsets.

SP/1	: expressions and output
SP/2	: variables, assignment, and input
SP/3	: selection and repetition
SP/4	: character strings
SP/5	: arrays
SP/6	: procedures
SP/7	: formatted I/D
SP/8	: records and files

Figure 6.2 SP/k Subsets

One of the great advantages of a system such as this is that the student will not become confused by the accidental use of constructs they do not yet understand: they simply do not exist. Choosing a programming language is usually a difficult decision. It is usually desired to choose a language that is used outside of the university environment, yet it must also be straight-forward enough for first-year students to grasp, and flexible enough to illustrate more advanced concepts as the students gain experience. Quite often, this need is filled by the use of several languages. Although most would agree that students should be exposed to several different programming languages, it is also important that the students have the opportunity to deal with new concepts using a language they are thoroughly familiar with, so that the peculiarities of

the language don't clutter up the main issue.

Plato

Plato is probably one of the best known 'Computer Aided Instruction' systems available today. It was developed at the Computer Education Research Laboratory (CERL) at the University of Illinois. It was intended to be a cost-effective, general purpose computer-based educational system [Lyma77]. Although used primarily for secondary school, it also has potential uses in a university environment — for drill and practice, problems, presentations and simulated experiments. Plato uses specially designed intelligent terminals that allow the addition of numerous peripheral devices (audio devices, music synthesizers, etc.). The system itself (the one used by CERL, other configurations may vary) is backed up by two CDC machines with two million words of extended core, and uses a programming language ('Tutor') created to minimize the need for specialized knowledge of programming in order to create courseware.

Teachers who have used the system feel it to be helpful [Rant80]. Students are more motivated and therefore work longer. There are, however, several problems with a system such as this: it is still quite expensive to install, the course-writing languages are still too complicated to make them truly useful, and in order to use Plato, one must still tie in to a large system [Hall80]. Some of these problems are being remedied, and versions of Plato will be available on some microcomputers.

Logo and Karel the Robot

As the two systems, Logo and Karel the Robot are similar in many respects, they will be discussed together.

Logo [Pape80] provides an environment intended to help children learn to use computers. Users of Logo write statements that operate on an object called a 'turtle'. The 'turtle' can be a physical object that moves or a symbol on a CRT that draws pictures on the screen. Papert refers to them as 'objects to think with'. Logo is based on Lisp: it is highly interactive; it is interpretive, and allows procedural definitions, with facilities for both local variables and recursion. The 'turtle' can move a specified number of 'steps', turn a given number of degrees, put its pen down (start drawing),

and lift its pen up (stop drawing). In addition to these basic capacities, the system has many more advanced capabilities (arithmetic, file operations, etc.).

Karel the Robot [Patt81] is also designed to help people learn to use computers, but the audience is the university student rather than children. The system (along with the textbook) is designed to be covered in the first few weeks of an introductory programming course. Karel's world consists of a city that has streets and avenues along which Karel can move. Some streets and avenues may have walls between them which Karel cannot cross. Karel has the ability to move in the direction he is pointing (one block at a time) and to turn left 90 degrees. He can also pick up and deposit objects called 'beepers'.

The Karel simulator is written in Pascal, and Karel's language is based on Pascal. The language has no actual variables or data structures, but it has the begin-end construct, as well as most of the major control structures of Pascal. Like Logo, Karel is also interactive and interpretive. It allows procedural definitions, including recursive ones (control can be implemented using 'if', 'while', or 'iterate' statements).

Both of these systems have many points in their favour. They allow the users, be they children or adults, to get started almost immediately. Because they are so interactive and graphic, the users get immediate and easily understandable feedback. These systems are fun — gamelike — therefore users are likely to become more involved. The systems are also essentially non-numerical: this avoids the 'fear of math' that causes a block in many users. Using either of these systems, one can learn the basic principles of programming quickly and in an environment that is isolated from the other concepts a programmer needs (it becomes possible to separate algorithms from I/O variables, compilation).

These systems have numerous qualities that make them useful for teaching problem solving skills. Users learn to break objects and tasks down and describe them in parts. Perhaps most importantly, the users have immediately visible, recognizable applications with which to work (the applications have a connection with something they know). Users can achieve success very quickly, and then build on those 'procedures' to solve larger and larger problems. Conversely, they can quite easily see how a larger problem can be broken down into parts they already know how to solve. The results of their efforts (the output) are dynamic: this makes it easier for students to

visualize solutions (they can imagine what Karel or the turtle must do).

Although Logo has several advantages over Karel when viewed in terms of their appropriateness for post-secondary teaching, Karel seems the better system. Logo allows the user to draw pictures — a very concrete goal — which is personally gratifying. Logo is also available on many systems both large and small for as little as \$150.00 [Will82].

Karel, on the other hand has the substantial advantages that the intended target is that for which it is to be used, as well as the fact that the teaching materials have already been produced (textbooks and exercises). Karel has fewer primitives than Logo so there is less to remember. Karel is also much more like the language they are about to use than is Logo — this makes the step to Pascal relatively painless. Also because of its similarity to Pascal, Karel goes a long way towards promoting good programming habits (more so than the unstructured nature of Lisp) and the textbook attempts to teach structured programming habits.

6.2 Hardware Tools

As a general rule, fewer hardware than software tools exist. Most of those that are available, however, have the potential for being very useful.

Shift Calculators

Although by today's standards, a shift calculator would not be classed as hardware, it is included largely for historical reasons. As tools, they are not longer used for their original purpose, but they do have one very important application. They provide an excellent vehicle for illustrating how low level multiply and divide algorithms (logic design) work. For multiplication one number is entered, and the user must turn the crank the appropriate number of times in each place ('10's, '100's, etc.) before shifting to the next one. For division, the crank is turned the other way until the number becomes negative, then it is turned back once, and that count is recorded before shifting to the next place. No formal studies have been done, but the students who have used them seemed to enjoy them and generally felt quite positive.

DEC Logic Labs

In Chapter 5, it was found that some instructors were dissatisfied with the treatment of digital circuits. Although now quite old, the DEC Logic Labs still serve well to show the composition and operation of simple circuits. Students can build these circuits and observe their behavior.

MicroProfessor

Ideally, to learn important concepts in computer operation, one wants a stand-alone system that the students can use and with which they can experiment (without the danger of interfering with other people's processes). MicroProfessor is such a system. It is quite new (1981) and is available from Multitech Electronic Inc. It was designed specifically to teach microprocessor concepts and the fundamentals of microcomputer operation. The system is composed of a Z80 processor chip with on-board RAM and ROM; it has a 36-key keyboard, and an internal power supply. One can load programs, set breakpoints, and single-step through programs. There are also various additional devices that can be attached (including tape recorders, printers, and a speech synthesizer). The system comes complete with three manuals: one for general reference (User's Manual) which contains examples; an Experiment Manual containing 13 complete experiments; and the monitor program source listing. A student workbook is also available, though not included.

This system has many advantages. It is very inexpensive (the basic system complete with the three manuals is available for approximately \$150.00) and has been designed specifically for teaching purposes. In fact, it is actually designed to be self teaching, so the documentation is especially clear. In addition, the hardware has been carefully laid out and has all been labeled. Finally, and perhaps most importantly, the system is uncluttered — it has been designed for a single purpose. Although it would not be impossible to write programs to accomplish personal tasks, it was not meant to be so versatile that the basic system becomes inaccessible.

6.3 Other Tools

There are some other tools that can be considered neither software nor hardware. As the discipline we are dealing with is the study of computers, it seems that many

instructors ignore the possibility of using tools that do not involve the computer (with the possible exception of films).

Towers of Hanoi

Recursion is a topic that every year confuses the students and aggravates the instructors. The difficulties in teaching this subject have already been discussed (Chapter 5). It seems unlikely that a single tool will suddenly make it all clear, but one attempt has been made at the University of Calgary. One instructor built models of the Towers of Hanoi (with five discs), which were used in the second year class. The students were allowed to manipulate these models during a lab period and were slowly guided towards a written solution to the recursive algorithm. They were asked questions, given time to answer, and then they were given the correct answers so they could move on to the next step. The students enjoyed the experience, and although many were still somewhat confused (about recursion), most agreed that having an actual model to work with helped them work out their algorithms.

CARDIAC Kits

The Cardiac Kit was designed by Bell Telephone Labs in 1968 to help people understand the basic operation of a computer (hardware systems organization), 'Cardiac' is an acronym for 'CARDboard Illustration Aid to Computation'. As the name implies, the Cardiac Kit is made completely of cardboard. It uses a simplistic hypothetical machine language which 'runs' on the cardboard model. It was designed specifically for teaching — in fact, it has no other use. The Cardiac Kit comes with a booklet that has clear explanations and numerous examples. The booklet also directly addresses the problem of 'bootstrapping', a topic with which several instructors were dissatisfied.

Although this tool is now quite old (in fact, it is now almost as old as the next set of first year students!), it still fulfills the purpose it was originally designed for quite admirably. It is a simple, straight-forward method of introducing students to the basic concepts and ideas of computer operation.

Sorting Out Sorting

Films are not used as much in computer science as they are in some other disciplines but they do deserve recognition, and several excellent examples are currently available. 'Sorting Out Sorting', produced by the University of Toronto is one of these. Through the use of animation, it describes and illustrates nine algorithms for three different types of sort (linear, exchange, and selection). Each algorithm is illustrated and described by means of lines of varying lengths (items to be sorted) that move to the position described by the algorithms. Three algorithms in each group are described, then compared to each other (via speed comparisons) before going on to the next group. The visual effect of the actual sorting behavior is quite dramatic, and illustrates the point far better than tracing the execution of a program on the blackboard and listing the times for completion. After all of the algorithms have been introduced, all nine are compared with each other — with interesting effects. At the end of the film, the whole film is repeated in a fraction of its original time — an extremely useful strategy — which refreshes the students' memories and helps them remember what they have seen. The whole film takes approximately 30 minutes and is most definitely a far more effective use of that time than any purely verbal explanation could be.

Although there is still much room for improvement in the area of teaching tools for computer science and many areas of need that would definitely benefit from such attentions, numerous well-designed and thoughtful tools do exist, but sadly, few actually use them. Practical problems prevent the widespread use of some. For example, although a device such as the MicroProfessor is a little too expensive to expect all students to buy their own, problems with theft and vandalism prevent some departments from using them in their classes. In addition, they are understandably unwilling to complicate the already heavily taxed administrative staff's duties by expecting them to manage loans with or without safety deposits. Perhaps when the institutions begin to realize that computer science departments are unique in their requirements, allowance can be made to provide for adequate staff and useful tools will begin to make a more prominent appearance.

The problem does not only lie with administration, however. With a bit of forethought on the part of the instructors, many of the tools can be made available to the

students. Instructors are seldom given encouragement to organize their classes or develop tools of their own. In fact, as most are also required to do research, many are discouraged from becoming effective teachers. Sadly, there are even those who feel that teaching should not be their job at all, forgetting that research cannot be advanced if no one has been taught to do that research.

Chapter 7

A FEW ADDITIONS

The preceding chapters have dealt with the need for teaching tools in computer science and the last chapter introduced a few that are currently available. According to the results of the questionnaire (chapters 4 & 5), there are numerous areas where many instructors felt the treatment was less than satisfactory (see Table 5.1). These are areas that could probably benefit most from the development of new tools. This chapter discusses three new tools, developed at the University of Calgary to fulfill various needs. These tools have been implemented and more information is available upon request.

7.1 Syntax Diagrams

The syntax of a language is usually considered to be the easiest part to learn; still, introductory programming students spend a disproportionate amount of time on syntax (learning, looking things up, debugging). This situation can be improved somewhat with the use of the following tool.

Many introductory Pascal textbooks use syntax diagrams to introduce new constructs. Quite often, those textbooks that use the syntax diagrams make no attempt to explain them (they can be useful). Students can be taught to use these syntax diagrams effectively. By using the syntax diagrams, students can make sure their own programs are syntactically correct and they can quickly locate syntax errors when they do occur.

Two types of exercise have been developed in November of 1982 [Beck83] to allow students to learn the syntax of the programming language they are going to use (in this case Pascal).

The object of the first set of exercises is to reduce given samples of Pascal code, beginning with the simple tokens and ending with the highest level syntactic unit for that sample. The syntax diagrams prescribe the rules for reduction.

The student must make several passes over the sample, reducing the tokens to the next higher set in each pass (each pass is written down).

```

    if i = 10
    then begin
        i := 0;
        x := x * x
    end
    else begin
        i := i + 1;
        x := y
    end

```

PASS 1:	PASS 2:	PASS 3:	PASS 4:
if E then begin ID := E; end else begin ID := E; end end	if E then begin S; S end else begin S; S end end;	if E then S then S;	S;

Figure 7.1 Example — Exercise 1

The second set of exercises also uses the syntax diagrams, but in this case a list of syntax error messages is also required. Here the student is asked 'to play compiler', and must parse the samples. Each sample specifies the starting unit (eg. program or if statement), and, with this, the student is to follow the syntax diagrams, expanding each syntactical entity appropriately until they are able to follow the terminal symbols as they appear in the sample. When an error is found (a terminal symbol in the sample that does not match with an alternative from the syntax diagrams), they

search the list of error messages, record the most appropriate one and follow the attached instructions for recovery to continue.

```

program one (input, output):
var    1 : integer;
begin
    write ('Number please: ');
    while not eof( input ) do
    begin
        read (i)
        writeln ('That was a', i)
        write ('Number please: ')
    end
end.

```

RESULT AFTER PARSING:

```

P[ PH[ program id ( id, id ) ] ;
  B [ VD [ var id : t ] ;
    begin
      S1 [ id ( CS[ 'characters' ] ) ]
      S5 [ while E do
        S2[ begin
**ERROR 12 ;      S1[ id ( CS[ 'characters' ], id ) ]
**ERROR 12 ;      S1[ id ( CS[ 'characters' ] ) ]
          end ]
        ]
      end ]
    . ]

```

Figure 7.2 Example — Exercise 2

These exercises have been tested on a second-year computer science class and the comments were favorable. The class used the exercises to help them review Pascal and most agreed that they could be useful in finding bugs in programs.

With some refinement, these syntax diagrams can be used in the first year when students are learning their first language and hopefully, it will make them more aware of the syntax as they begin to write their programs.

7.2 Microtool

Microprogramming is a topic that few instructors said they covered in the introductory curriculum. Many seem to feel that this topic is too advanced for most introductory students, but this need not be the case. Jim Parker at the University of Calgary has developed a software tool to deal with microprogramming [Park83b].

When teaching about microprogramming, there are certain notions of importance. One is the concept of a machine inside a machine, and another is the notion of the machine interpreting instructions to it. Most conventional methods of teaching microprogramming fall into two categories. First is the blackboard and textbook treatment of a simplified example system (often accompanied by a programming assignment to simulate the (macro) computer's operation), and the second is the use of a commercial system with writable microstore. With the first method, the students do not have the opportunity to experience the behavior of a real system, and the second is rather expensive (of both money and space) and sometimes difficult to manage.

The system developed by Parker is based on the processor described by Tanenbaum [Tane76]. This design was chosen primarily because of the availability of a relatively popular text that could be used in conjunction with the system. The 'machine' uses a 40-bit microinstruction, and the system permits users to enter microcode from the terminal. It is also possible to load machine code from files. The user can step through the execution of programs, set breakpoints, and the system allows tracing and dumping of both 'macro' store and the control store.

Among the advantages of this system over the other methods is that students get practice with writing and executing microprogrammed instructions without the complexities of dealing with a real machine. The students have access to high level debugging tools and can create microprograms using higher level editors. This way, students can concentrate on the concepts being taught without having to fuss with all of the 'extras' found in actual machines. The system is reasonably portable (it is written in Pascal and runs on a VAX UNIX system), but perhaps most importantly, it allows

large numbers of students to have access without the danger of harming the processor on which it runs.

It was not possible to test the system on students to get their reactions, but it was still possible to get some comments from other instructors. Of primary concern was the design of the 'machine' itself. It was felt that although it might make a good starting point for the students, it was too simplistic to be useful for long. To remedy this situation, it would become necessary to alter Tanenbaum's design, thus making the textbook somewhat less useful. For the most part though, the reaction was positive and it is felt that such a system would help make the previously mentioned concepts a little easier for the students to grasp.

7.3 Finite State Machine Simulator

Those that answered section 8 of the questionnaire ('Theory of Computation') were basically satisfied with the results of their efforts. Theory is an area that involves many abstract concepts and one that many students find very difficult indeed. Few tools are available that instructors can make use of besides the traditional ones (textbooks, lectures, etc). Because of its inherent difficulty, it is unlikely that the development of new tools will make this subject simpler, but it is felt that the employment of a few tools that allow the students to observe and work with some of the many dynamic processes involved might make some of the concepts a little less abstract, and thereby somewhat easier to grasp.

The tool that has been developed is intended to help students understand finite state machines and how they operate. Currently, this tool is a software program that runs on a VAX (under UNIX), but this simulator is basically just a prototype for a hardware box to be built specifically to demonstrate finite state machines. The original idea was put forth by John Slater, but has since been expanded.

The purpose of the simulator is to provide a vehicle for manipulating sample instances of finite automata. The simulator contains numerous descriptions of finite automata which can be displayed on request. It also allows the user to define strings and test them to see if the FA will accept them. Once a new string has been entered, the user is to indicate which state is to be used as the initial state, and then each time the user presses the return key, the finite state machine takes the next character as

input and uses it to move to the next possible state.

By working through various examples, it is hoped that the user will gain experience with finite state automata and their significance. When the simulator is first invoked, a blank Finite State Machine (FSM) is drawn on the right side of the screen and the user is prompted for a command. The prompt (' >') appears along the bottom of the screen. The commands are all entered as single letters (except one) followed by a <return>. The exception to this is 'pulse' which is a <return> character all by itself. A brief explanation of each command is found in Figure 7.3

l	(load) :	load a new FSM
w	(word) :	define a new string
< return >	(pulse) :	move to next state based on input
i	(ignore) :	skip current character
r	(reset) :	reset the error flags
s	(state) :	move to specified state (used to resolve non-deterministic input)
c	(clear) :	erase the current string
q	(quit) :	leave simulator

Figure 7.3 Commands for Finite State Machine Simulator

The Finite State Machine Simulator contains descriptions for a number of different finite automata. The user may request any one of them by loading it into the state table (using 'l'). The F.A.'s are referred to by number. Once loaded, the graph is displayed on the right-hand side of the screen along with the definition, which is described on the left-hand side. The user may then create strings/words (using 'w') and observe the behavior of the FSM as the characters are processed (using 'pulse'). If the string is part of the described language, it will be possible to 'pulse' to the end of the input, at which point the user sees a message that indicates the end has been reached. Under various conditions, a number of error flags may be set while processing a given string. These error flags indicate non-deterministic states, symbols encountered that are not members of the alphabet, and attempts to move from a state using a symbol not allowed for that state.

This simulator was given to a second year class at the University of Calgary that was studying finite automata at the time. As rigorous testing was not attempted (see discussion at end of chapter), the students were merely asked to try the system and return their comments. Most students liked the idea, although they felt the system was not flexible enough. They wanted to be able to define their own automata. This is a capability that the final system will definitely have, along with more possible states (the current system has five), and access to the full keyboard for the alphabets. Even though the graphics were found to be somewhat confusing, the students thought the system was fun to work with and many commented that the system helped them to visualize other automata with which they were working.

Although none of the comments given in this chapter can be used to prove conclusively the utility of the tools described, it is fairly clear that most students appreciate the efforts and seem to feel that the opportunity for 'hands-on' experience is a valuable one.

Rigorous testing was not used with any of these tools for several reasons. This type of testing is exceedingly difficult to control properly and there is the danger that the statistical results, although significant, may be misleading (they may indicate that something is significant, but the reasons may be contrary to what the researcher assumed). In addition, there is a question of ethics to be considered. These tools are thought to be potentially very useful, and initial reactions bear this out. As a result, it could be considered to be unfair to allow some students the benefits of these tools and not others. It is not within the scope of this thesis to try to solve these problems. Hopefully, it is sufficient to state that the students and instructors who reviewed the tools outlined in this chapter would indicate that further work in the area of tool development would be welcomed by all involved.

Chapter 8

NEW TOOLS

In the last chapter several tools that have been developed recently were described and the reactions of the students who used them were discussed. According to the comments of those who answered the questionnaire (chapter 5), there are numerous areas where instructors felt the treatment was less than satisfactory (see Table 5.1). These areas of dissatisfaction are areas that would probably benefit most from the development of new tools. This chapter deals with three such tools. None of these tools has been implemented as yet, but the initial designs are complete and these will be discussed in the following paragraphs.

8.1 Recursion

The first of these proposed tools was originally designed at the University of Calgary [Park83c]. It deals with recursion, which has traditionally been a very difficult concept to master. Often, even when the students understand individual applications, they still do not grasp the general concept.

Finding suitable examples is difficult. The students often have difficulty following the example itself, so the point being made by it is lost. Recursion is a very dynamic process and most techniques used are rather static (blackboard drawings, textbooks, etc.). To deal effectively with a dynamic process, it is felt that the tools used should also be dynamic.

Parker has suggested that a film be produced to illustrate the concept of recursion. A film would be an ideal medium, in which the powers of animation and 'trick photography' could be used to great advantage. One example that can be illustrated in a film is a recursive tree traversal. A recursive tree traversal can be shown using both a representation of the tree (diagrammatic) and the code for the procedure being executed. Each new invocation would result in the procedure call being expanded and the tree being modified such that the part of the tree that is accessible by the new invocation is a different colour from the rest.

Another technique that would be employed by the film that would be quite illustrative is a 'mock news cast', similar to what is seen on television. Frequently, during a typical newscast, the anchor-man will refer to another reporter for a more detailed account of a story. When the reporter has finished speaking, he or she will refer back to the anchor-man who continues with the broadcast. Occasionally, the reporter will go a level deeper and refer to yet another reporter. This technique can be used quite effectively to illustrate how recursion works. To make the example approximate recursion a little more closely, the same announcer would be used for all parts. Instead of referring to another reporter, the anchor-man would refer to himself. This could be carried to several levels and upon returning, each 'invocation' would conclude in the same manner as every other, and then refer back to the previous 'call' (" . . .and now back to you Fred . . .").

Another device that can be used to advantage to illustrate recursion in the film is the mathematical puzzle. Several mathematical puzzles exist whose solutions rely on principles of recursion (such as 'Computer Loops'). These can be solved and their solutions shown using animation (in this case 'trick' photography) so that no-one's hands interfere with the observation of the solution.

Although no single tool is likely to suddenly make a difficult concept like recursion easy to learn, the use of a few carefully selected techniques can make the learning process a little more enjoyable and the students a little likelier to grasp the ideas faster.

8.2 Floating Point Demonstrator

Another area that instructors were dissatisfied with is floating point representation and arithmetic (part of logic design). As stated in chapter 5, these topics are primarily factual, thus it is likely that demonstrations and exercises will be of benefit. One way to accomplish this is with the use of a software package. A drill and practise type of 'CAI' program could be quite useful. It is possible to display various representations (perhaps those that are used in the machines available to the students), present material, and ask questions that can be monitored by the machine. The students can observe simple floating point arithmetic (more thorough treatment could perhaps be left until the senior years).

The system can display numbers in three formats: decimal, octal, and binary. The decimal format can be used initially so the students have a number system with which they are thoroughly familiar, to be used until they understand the basic principles. Then they move on to octal and binary numbers.

To learn about the representation and normalization, students are first shown how various numbers (decimal) are converted to floating point and what happens during normalization. They are then prompted with numbers chosen by the system and asked to enter the floating point representation in the normalized and non-normalized forms. Learning about floating point arithmetic is accomplished in the same manner. The students are shown the algorithms for performing addition, multiplication, etc. in a stepwise fashion and later asked to follow through by themselves (entering intermediate results).

The system itself need not be very elaborate; numerous lessons preceded by explanations and succeeded by questions or a quiz may be quite adequate. Although quite important, floating point is not as troublesome a topic as, for example, recursion or problem solving and as such, it is felt that somewhat more exposure and a little more practice would help to bring the ideas across.

8.3 Treetool

In the area of data structures, 'lists' was a topic where the coverage was found to be somewhat lacking. Manipulation of data structures, as so many other topics in computer science, is very dynamic, and as a result it is difficult to show using static tools (eg. blackboards, overheads, etc.).

Quite often, students find that programs written to implement algorithms that manipulate data structures are difficult to debug. The students may have a vague 'picture' of the algorithm but cannot imagine what might be causing the bugs when they appear. One substantial problem with debugging these programs is the lack of diagnostic output — especially when pointers are involved; 'pointers' is a concept that many students find problematic in the first place.

In order to help teach these concepts, we need a tool that allows students to manipulate various data structures without the cluttering effect of a complete language and with the ability to observe the structures as they are manipulated (observe the

effects of the algorithms as they are carried out). One such tool is the 'treetool'. This tool deals specifically with binary trees, primarily to keep the design simple (it would be possible to create a general data structures system along the same lines).

The tool is a software system that the students use to help them answer questions that they are given and to observe and implement various algorithms which manipulate binary trees. The student uses predefined nodes to perform the common operations on binary trees — namely: insertion, deletion, and traversal. The 'language' that is used to do this is a subset of a known language (Pascal) to make it easier for the students to begin using the system quickly. All required variables are pre-defined as well as the procedure interfaces (via procedure headings). This is intended to allow the students to concentrate on the actual algorithms as much as possible. Also in an effort to keep the point of the exercises as obvious as possible the allowable operations of the language are severely restricted. The student is allowed to assign values to the variables, print the values of keys, and call any of the three procedures. To make the system complete for dealing with tree structures, there exist 'if' statements and a 'while' statement.

The system is intended to be highly interactive, so many actions can be performed from 'command mode'. Among these actions are: the ability to interrupt and resume execution; the ability to execute one of the three procedures; the ability to create a new tree to work with and to perform various other bookkeeping chores. It is also possible to define a new version of any of the three procedures.

The system uses the terminal screen in two ways. One part of the screen is used to display the currently active procedure (the code itself is displayed along with a marker pointing to the currently active statement). A small part of this half of the screen is also used for the output from the routine and for echoing the most recently issued command. The other half of the screen is used to display the tree itself. Markers are used to show the current values of the variables. These markers are changed when the value of the variable changes (as does the tree when appropriate). For a complete list of commands and description of the language, refer to tables 8.1 and 8.2.

The students using this system are provided with an explanatory manual about the system and a set of exercises they are to complete. For example:

Create a tree with the following nodes: 2 6 12 9 5 4
 Where in the resultant tree would the node with the key '7' be inserted?
 Draw the tree and check your answer by inserting it in the tree shown.

The exercises are intended to guide the student through a series of demonstrations, and present to the student various questions along the way. With a system such as this, the students watch what is happening to the variables and the tree as the code is executed and they can examine the results of certain actions. This will hopefully help build an intuitive 'picture' of what the routines are doing. It also allows the student to actually observe the effect of various bugs as they occur.

Although not yet implemented, this system is intended to be completed in the upcoming academic year and will be tested on a second year class at the University of Calgary (Cpsc 302). The system will initially deal only with binary trees, but, as

Table 8.1 Treetool Commands

ctrl S	: PAUSE: halt execution
c	: CONTINUE: start execution (pick up where 'pause' left off)
proc 'X'	: DEFINE PROCEDURE: start input for subroutine 'X' ('X' can be one of traverse, delete, or insert)
traverse	: CALL TRAVERSE: call traverse routine with main root
insert N	: CALL INSERT: call insert routine with N (insert new node with key = N)
delete N	: CALL DELETE: call delete routine with N (delete node with key = N)
restore 'X'	: RESTORE: restore standard subroutine 'X' ('X' can be traverse, delete, or insert)
stop	: CLEANUP: get rid of information created by 'pause' for 'continue'
make N1 N2	: MAKETREE: build a standard search tree from the given input (maximum number of keys is 15)
clear	: CLEARTREE: delete tree, set root to nil
quit	: QUIT: leave treetool system
N 'statement'	: CHANGE: change line N to the statement given
display 'X'	: DISPLAY: display the named routine on the screen ('X' can be traverse, insert, or delete)

Table 8.2 Treetool Language Definition

PRIMITIVES: const nil = 0;
 type node—ptr = 0, , 15;
 node = record key : integer;
 left : node—ptr;
 right : node—ptr;
 end;
VARIABLES: var root : node—ptr;
 p : node—ptr;
 n : node—ptr;
 flag : boolean;
 k : integer;
KEYWORDS: if then else begin end while
BUILTINS: print (< integer >) :print value as output
 new (node) :return new node
 return pointer to it
SPECIAL SYMBOLS: := () , ; . ^ = < = > < >
SUBROUTINE HEADERS: traverse (var root: node—ptr);
 insert (var k: integer,
 var root: node—ptr);
 delete (var k: integer,
 var root: node—ptr);

SYNTAX

Integer Value < ival > :: = < integer > : < iref >
Integer Reference < iref > :: = k | < locator > . key
Pointer Value < pval > : : = new (node) | < pref >
Pointer Reference < pref > : : = root | p | n |
 < locator > . left | < locator > . right
Boolean Value < boolval > : : = true | false | < boolref >
Boolean Reference < boolref > : : = flag | < ival > < = < ival > |
 < ival > > < ival > | < ival > = < ival > |
 < ival > < > < ival > | < pval > = < pval > |
 < pval > < > < pval >
Locator < locator > : : = < pref > ^
Assignment < assign > : : = < pref > : = < pval > |
 < iref > : = < ival > |
 flag : = < boolval >
Print < print > : : = print (< ival >)
If — Then — Else < if > : : = if < boolref > then < s > |
 if < boolref > then < s > else < s >

Table 8.2 (continued)

```

While <while> : := while <boolref> <s>
Call <call> : := traverse ( <pref> ) |
                    insert ( <ival>, <pref> ) |
                    delete ( <ival>, <pref> )
Statement <s> : := <empty> | <assign> | <print> | <if> |
                  <call> | <while> |
                  begin <s> {; <s> } end
Procedure Finish <pf> : := end.

```

stated earlier in this chapter, it can be expanded to include other data structures as well (eg. linked lists, matrices, queues, stacks, etc.).

It seems certain that the development of new tools cannot either replace the role of the instructor of a course nor make the task of learning computer science a simple one, but hopefully they can serve both the students and the instructors in useful ways. Through the use of carefully designed tools the student can be better motivated and the task of learning can be made somewhat more enjoyable. For the instructor, the benefits include somewhat less time spent on preparation which can be very important when most have very little time to spare.

Chapter 9

CONCLUSIONS

The list of topics and objectives given in the appendix is one of the major outcomes of this thesis. It outlines the knowledge and abilities that computer science students should have upon completion of the introductory part (approximately the first two years) of their programs. While several other groups and individuals list topics to be included in such a curriculum, most give no indication of what they mean by a particular topic (one person's interpretation of a particular topic may be vastly different from another). These curricula also do not state any goals to go with the individual topics. It is very important to outline goals when discussing a curriculum. Aside from defining the topic more completely, it also gives a better indication to the instructors in a computer science department as to what they should be attempting. These guides should not be too rigid, however. Instructors must be given a reasonable amount of freedom to develop their own courses. By using the topics and objectives as a basis to work from, it is possible to build a curriculum that is both consistent and complete.

A list of topics and objectives alone is not sufficient. It is also necessary to know how these topics are interrelated. This has also been recorded in this document using the previously mentioned list as a basis. The description of the interrelationships in the third chapter is not complete and open for debate. It is hoped that more work will be done in this area to define more fully the interdependencies of the subjects taught in computer science. Perhaps, some topics that have proven very difficult for the students may be somewhat easier if presented in a different order.

In order to validate the list of topics and objectives, it was given to faculty members of computer science departments in most Western Canadian universities so they could comment on which topics they covered. The results of the questionnaire have been described and recommendations laid out. In particular, those topics with which instructors were displeased were singled out and possible solutions suggested. Primary attention was paid to the tools used in teaching. Numerous tools currently available were reviewed and then several new tools developed at the University of

Calgary were described.

The tools and techniques most often used were fairly traditional ones (lectures, blackboards, overheads, etc.); none of which are very dynamic in nature. So many of the concepts taught in computer science are very dynamic, yet most of the tools used are quite static. The importance of using dynamic tools to illustrate dynamic ideas cannot be stressed too strongly. The use of tools to fit the topic are likely not only to make the topics easier to learn, but also easier to teach.

Finally, three proposed new tools were described. These have not yet been implemented but will hopefully be completed in the coming year. In addition to this, the three already implemented will be revised and improved.

There are still numerous areas where instructors were displeased and thus many areas where the development of new tools could be of great value. Although it is possible that the use of too many tools can cause students to become as bored or confused as the use of too few, it is fairly clear that it will be quite some time before the danger of too many tools becomes real.

REFERENCES

- [Abel82] Abelson, Harold, "A Beginner's Guide to Logo — Logo is not just for kids". BYTE Vol. 7 No. 8, Aug, 1982, pp. 88-112.
- [Aust79] Austig, Richard H., Bruce H. Barnes, Della T. Bonnette, Gerald L. Engel, Gordon Stokes, Editors, "Curriculum '78: Recommendations for the Undergraduate Program in Computer Science — A Report of the ACM Curriculum Committee on Computer Science" CACM Vol. 22 No. 3, March, 1979, pp. 147-165.
- [Beck83] Becker, K., "Teaching Syntax in an Introductory Programming Course", Proceedings of Converging Technologies Conference, Canadian Information Processing Society, Ottawa, Ont. May 16-20, 1983, pp. 183-195.
- [Bram83] Bramwell, Bob, "An Automatic Manual", MSc Thesis, University of Calgary, 1983.
- [Chan77] Chandor, Anthony, John Graham, and Robin Williamson, "The Penguin Dictionary of Computers", Second Edition, Great Britain, Penguin Books Ltd., 1977.
- [Coug73] Couger, J. Daniel, Ed., "Curriculum Recommendations for Undergraduate Programs in Information Systems", CACM, Vol. 16, No. 12, Dec. 1973.
- [CSRG81] Computer Systems Research Group, "Sorting Out Sorting", (film), Dynamic Graphics Project Production, University of Toronto, 1981.
- [C3S 68] Curriculum Committee on Computer Science (C3S), "Curriculum '68, Recommendations for Academic Programs in Computer Science", Comm.ACM 11,3 (March, 1968), pp. 151-197.
- [Enge78] Engel, Gerald L., and Oscar N. Garcia, "Curricula Development in Computer Science and Engineering", IEEE Trans. Ed., Vol. E21, No. 4, Nov. 1978.
- [Good80] Good, Thomas L. and Jere E. Brophy, "Educational Psychology: A Realistic Approach", 2nd Ed. New York: Holt, Rinehart and Wilson, 1980.

REFERENCES (continued)

- [Hall80] Hallworth, H.J., and Ann Brebner, "Computer Assisted Instruction in Schools: Achievements, Present Developments, and Projections for the Future", Planning and Research, Alberta Education, June, 1980.
- [Harv82] Harvey, Brian, "Why Logo? : Logo is designed to encourage development of problem-solving skills", BYTE, Vol. 7, No. 8, Aug., 1982, pp. 163-193.
- [Holt77] Holt, R.C., D.B. Wortman, D.T. Barnard and J.R. Cordy, "SP/k: A System for Teaching Computer Programming", CACM, Vol. 20, No. 5, May, 1977. pp. 301-309.
- [IEEE77] IEEE Education Committee (Model Curriculum Subcommittee of the IEEE Computer Society), "A Curriculum in Computer Science and Engineering", Committee Report IEEE Publication EH0119-8, Jan. 1977.
- [Kern79] Kernighan, Brian W., and Michael E. Lesk, "LEARN — Computer Aided Instruction on UNIX", Bell Labs Publications, January 30, 1979.
- [Lisk74] Liskov, Barbara, and Stephen Zilles, "Programming with Abstract Date Types", SIGPLAN Notices, Volume 9, No. 4 (April, 1974), pp. 50-59.
- [Lyma77] Lyman, Elizabeth R., "PLATO Makes Learning Mickey Mouse", ROM. Sept. 1977, pp. 34-39.
- [Muld75] Mulder, Michael C., "Model Curricula for Four Year Computer Science and Engineering Programs: Bridging the Tar Pit", Computer, Dec. 1975.
- [Pape80] Papert, Seymour, "MINDSTORMS: Children, Computers and Powerful Ideas", New York: Basic Books Inc. Publishers, 1980.
- [Park83a] Parker, J.R., Personal communication, January and February, 1983.
- [Park83b] Parker, J.R., "The Microtool Processor Emulation System", University of Calgary Yellow Series Report No. 82/110/29.
- [Park83c] Parker, J.R., ". . . And Now, Back to You", A preliminary screenplay for a movie about recursion. In preparation, 1983.
- [Patt81] Pattis, Richard E., "Karel the Robot: A Gentle Introduction to the Art of Programming", New York: John Wiley & Sons, 1981.

REFERENCES (continued)

- [Poly57] Polya, G., "How to Solve It: A New Aspect of Mathematical Method", 2nd Ed., Princeton, New Jersey: Princeton Press, 1957.
- [Rant80] Rantanen, J., "The Plato Experiment in Finland", Teaching Informatics Courses, HLWJackson (Editor) North-Holland Publishing Co., IFIP, 1982, pp. 75-86.
- [Sipp76] Sippl, Charles J., "Data Communications Dictionary", New York: Van Nostrand Reinhold Company, 1976.
- [Skem79] Skemp, Richard R., "Intelligence, Learning and Action: A Foundation for Theory and Practice in Education", Chichester: John Wiley and Sons, 1979.
- [Slat82] Slater, J., Personal communication, November, 1982.
- [Tane76] Tanenbaum, Andrew S., "Structured Computer Organization", New Jersey: Prentice-Hall Inc., 1976.
- [Will82] Williams, Gregg, "Logo for the AppleII, the TI99/4A and the TRS80 Color Computer", BYTE, Vol. 7, No. 8, Aug. 1982, pp. 230-290.

TEXTBOOKS

1. Aho, Alfred V., John E. Hopcroft, and Jeffery D. Ullman, "Data Structures and Algorithms", Addison-Wesley, 1982.
2. Bohl, Maralyn, "Information Processing", SRA 1976, 1971.
3. Bradley, James, "File and Data Base Techniques", HRW Series in Computer Science, 1982.
4. Cooper, Doug, and Michael Clancy, "Oh. Pascal!", WWNorton and Company. 1982.
5. Eckhouse, Richard E., and L. Robert Morris, "Minicomputer Systems: Organization Programming and Applications (PDP-11) New Jersey: Prentice-Hall. 1979.
6. Engeler, Erwin, "Introduction to the Theory of Computation", New York: Academic Press, 1973.
7. Graham, Neill, "Introduction to Computer Science: A Structured Approach". West Publishing Co., 1982.
8. Haber, Fred, "An Introduction to Information and Communication Theory". Vol. 4. Advances in Modern Engineering, Mass: Addison-Wesley, 1974.
9. Hayes, John P., "Computer Architecture and Organization", McGraw-Hill Computer Science Series, 1978.
10. Hill, Frederick J., Gerald R. Peterson, "Digital Systems: Hardware Organization and Design", 2nd Ed. John Wiley and Sons, 1973, 1978.
11. Hopcroft, John E., and Jeffery D. Ullman, "Formal Languages and Their Relation to Automata", Addison-Wesley Series in Computer Science and Information Processing, 1969.
12. Hopcraft, John E., and Jeffery D. Ullman, "Introduction to Automata Theory. Languages, and Computation", Addison-Wesley Series in Computer Science and Information Processing, 1979.
13. Lippiatt, Arthur G., "The Architecture of Small Computer Systems", London: Prentice-Hall International Inc., 1978.
14. Lipschutz, Seymour, "Theory and Problems of Discrete Mathematics", Schaum's Outline Series in Mathematics, McGraw-Hill Book Co., 1976.
15. Mano, M. Morris, "Digital Logic and Computer Design", New Jersey: Prentice-Hall, 1979.
16. Mendelson, Elliott, "Theory and Problems of Boolean Algebra and Switching Circuits", Schaum's Vocational and Technical Series, McGraw-Hill Book Co.. 1970.
17. Pratt, Terrence W., "Programming Languages: Design and Implementation", New Jersey: Prentice-Hall, 1975.
18. Stone, Harold S., et al., "Introduction to Computer Architecture", SRA Computer Science Series, 1980.

TEXTBOOKS (continued)

19. Tokheim, Roger L., "Theory and Problems of Digital Principles", Schaum's Vocational and Technical Series, McGraw-Hill Book Co., 1980.
20. Washburn, Dale W., "Computer Programming: A Total Language Approach". Holt, Reinhart, and Winston, Inc., 1970.

Appendix 1

Topics and Objectives

1. INTRODUCTION TO COMPUTERS

AIMS: 1. Familiarity with computers: their capabilities, limitations
2. Understanding of computer function (operation) in general terms.

Components of the System (Hardware Configuration)

Objectives: 1. Draw schematically the different parts of the computer.
2. Describe how they are related and how they interact.

The CPU

objectives: 1. Give a general overview of what it does.
2. Identify what makes a computer different from a calculator (eg. stored instructions).

Memory/Storage

objectives: 1. List the types and uses of storage devices and data recording media.
2. Explain, in general terms, the operation of memory.

Input/Output Devices

objectives: 1. List the types and uses of I/O devices.
2. Describe how they interact with the rest of the system.

The Operating System

objectives: 1. Define the concept of an operating system.
2. Distinguish functions performed by O/S, program, hardware (in general terms).

Files

objectives: 1. Define the concept of a file.

2. OPERATIONAL USE OF THE COMPUTER

AIMS: 1. Familiarity with the use of the computer.
2. The ability to prepare and run programs.
3. Knowledge of how to find information, get help.

Terminology

objectives: 1. Define those terms the student is likely to come across when first learning to use the computer (eg. terminal, command, program, batch, data)

Keyboard Skills

objectives: 1. Proficiency at use of the keyboard (typing, knowledge of special characters, etc.).

System Organization

objectives: 1. Familiarity with the system that will be used
 2. Know how to access the system (logical - accounts, logging on, etc. and physical — terminals, printers, etc.)
 3. Diagram the system's file structure.
 4. List the important system facilities (libraries, text formatters, etc.)
 5. Describe available documentation (on- and off-line)

Commands

objectives: 1. Knowledge of function and result of the commands that the student will need.
 2. Knowledge of general command structure, types and uses of commands (eg. status commands, file manipulation commands, etc.)

Interactive Use

* Intended for those with access to interactive systems *
 objectives: 1. Familiarity with the use of an interactive system.

Terminal Operation

objectives: 1. Skills in terminal use — knowledge of special characters. terminal settings.

Logical Subsystems

Objectives: 1. Understanding of different levels of operation, how to move between them, escape, error recovery (eg. editors, debugging systems).

Batch Use

* Intended for those with access to batch systems *
 objectives: 1. Familiarity with the use of batch systems.

Keypunch Operation

objectives: 1. Skills in keypunch use — knowledge of special characters, keypunch settings, card formats, formatting, etc.

Running Programs

objectives: 1. Familiarity with the steps one must follow from the time the program is written on paper to the time it is handed in for marking (eg. entering the program, running it, preparing listings, etc.)

3. PROGRAMMING TECHNIQUES

AIMS: 1. An understanding of the steps involved in writing programs.
 2. Broaden experience with the problem solving process.
 3. Practice and experience in software design.
 4. Understanding the need for and familiarity with examples of different types of programming languages.

Problem Solving

objectives: 1. Formulate solutions to problems in a clear and concise manner.

Classification and Definition of Tasks

objectives: 1. Recognize various types of problems.
2. State problems clearly and completely.

Techniques: 1. Discuss paradigms of problem solving.

2. Knowledge of methods and approaches to problem solving (eg. top-down, 'divide and conquer', decision tables).

Algorithms

objectives: 1. Express solutions to problems in a manner that can be translated into a computer language with relative ease.

Concepts and Properties

objectives: 1. Define the concept of an 'algorithm'
2. Recognize types and characteristics of simple algorithms.

Expression of Algorithms

objectives: 1. Know how to express algorithms clearly (eg. drawing flow charts, pseudo-code)

Design

objectives: 1. Organize and define algorithms.
2. Discuss various programming methodologies (eg. data-flow analysis, top-down, bottom-up, abstract data types)

Analysis and Verification

objectives: 1. Determine whether the algorithms that have been written are complete and correct.

Programming Style

objectives: 1. Write programs that are clear, easy to read, understand and modify.
2. Evaluate the appropriateness of a language for implementing a given algorithm.
3. Discuss how the programming language affects one's approach to solving the problem.

Program Features

objectives: 1. Choose appropriate constructs in a given language.
2. Discuss the proper use of parameters and how they affect structure and design.

Readability

objectives: 1. Organize the program logically into modules.
2. Write programs such that the structure is obvious by appearance (eg. indentation, blank lines).
3. Know how to choose those constructs and statements that lead to maximum clarity (eg. while statements instead of counting loops with goto's, and meaningful variable names).

Documentation

objectives: 1. Make programs easily understandable by other programmers (eg. comments, external doc.).

2. Make program's function clear to the user (includes meaningful output, prompts, error messages, etc.)

Debugging and Verification

- objectives: 1. Demonstrate that a given program works.
 2. Find and correct errors in programs.
 3. Distinguish between run-time errors and compile-time errors.

Software Reliability

- objectives: 1. Describe what it means for a program to be reliable and robust.
 2. Describe measures for software reliability.
 3. Discuss testing of input data (how much; what is reasonable for a given application; what kinds of assumptions one might make)
 4. Discuss how system errors may be avoided or trapped (such as PL/1 conditions)

Error Detection and Correction

- objectives: 1. Understand the importance of documentation and programming style in error detection and correction.
 2. Describe numerous techniques for error detection (eg. reading cross-reference maps, etc.)

Programming Techniques

- objectives: 1. Employ various programming techniques ('Antibugging') to aid in writing correct programs (eg. explicit initialization)

Selection of Test Data

- objectives: 1. Select correct types and amount of test data for thorough checking (including pathological cases)

Debugging Software

- objectives: 1. Use available debugging software to find errors in programs (eg. interactive debuggers, setting break points, tracing programs)

Hand Execution of Programs

- objectives: 1. Skill in debugging programs through hand execution.

Abstract Data Types

- objectives: 1. Define the concept of an abstract data type
 2. List uses, advantages and disadvantages
 3. Describe how they might be implemented in several languages

Performance Evaluation and Efficiency Considerations

- objectives: 1. Describe efficiency in relation to programming and how some algorithms can be more efficient than others.
 2. Describe how efficiency may be measured.

Software Portability

- objectives: 1. Define software portability.
 2. Describe some ways it can be achieved.

System Dependencies

- objectives: 1. List several examples and state their impact (eg. word length, file access methods)

Language Dependencies

objectives: 1. List several examples and state their impact (eg. extensions)

Software Communication

objectives: 1. Write programs that share information (eg. files, external variables)
 2. Write programs that use separately compiled modules.
 3. Use external routines and variables, both system and user defined.

Numeric Computations

objectives: 1. Describe how expressions are evaluated and the importance of order and precedence.
 2. Describe what happens during assignment
 3. Discuss how operations affect accuracy

Manipulating Character Data

objectives: 1. State how strings can be manipulated and how assignment occurs.
 2. Describe the difference between fixed and varying length strings.
 3. Describe how the standard character codes (EBCDIC, ASCII, etc.) affect programming (eg. sorting alphanumeric data).
 4. Demonstrate how to convert between characters and numbers
 5. List the different kinds of operations that can be performed on strings (eg. concatenation, substrings)

Recursion

objectives: 1. Describe the concept of recursion.
 2. Recognize problems that lend themselves to recursive solutions.
 3. Recognize problems that should not be solved recursively.
 4. Implement recursive algorithms.

4. PROGRAMMING LANGUAGE STRUCTURE

AIMS: 1. Use one or two languages as examples.
 2. General understanding of how languages are built.
 3. Understanding of how compilers work, what they do.

Syntax and Semantics

objectives: 1. Define the terms.
 2. Distinguish one from the other.

Variables

objectives: 1. Define the concept of a variable.
 2. Define what a declaration is.
 3. Discuss defaults and how they affect variables.
 4. State how they are declared.
 5. Draw how they are represented.
 6. Describe how input and output is performed.
 7. Differentiate between computer variables and mathematical variables.

Types

objectives: 1. Define 'type'
 2. List the major types.

3. Differentiate between strong and weak typing.
4. Discuss how strong and weak typing affect programming style.

Calculations

- objectives: 1. Describe the characteristics of common arithmetic types (eg. integer, real) and how they interact.
2. Describe the effects of mixed mode arithmetic.

Statements

- objectives: 1. Distinguish different types of statements (declaration, conditionals, assignment, etc.)
2. Describe, in general terms how they operate.
 3. Discuss statements that alter the sequence of operations (call, goto. etc.)
 4. Discuss the various loop structures (while, for, etc.) and their uses.

Flow of Control

- objectives: 1. Describe mechanisms by which subroutines are called and how control is returned to the caller.

Scope Rules

- objectives: 1. Define 'scope'.
2. Describe the significance of scope rules.

Block Structure

- objectives: 1. Define 'block'.
2. Differentiate between local and global scope at all levels.

Static and Dynamic Scope

- objectives: 1. Define the difference.
2. Differentiate between lexically nested subroutines and nested subroutine calls.

System Scope

- objectives: 1. Identify elements of the system's scope (eg. builtins) and describe their effect on user defined elements.

Parameters

- objectives: 1. State how parameters differ from local and global entities.

Translators and Compilers

- objectives: 1. General understanding of what compilers are and how they work.
2. Contrast compilers and interpreters and define intermediate code.

Compiler Operation and Function

- objectives: 1. Describe the role of a compiler and what it accomplishes.

Syntax and Translation

- objectives: 1. Define the steps a compiler would go through during lexical analysis.
2. Differentiate compile-time errors from run-time errors (i.e. 'bugs' from syntax errors)

Formal Definition of Syntax

- objectives: 1. Describe, in general terms, how the syntax of a language can be defined (eg. BNF)
2. Use syntax diagrams, BNF, or some other formal system to describe

simple constructs and to determine whether examples of these constructs are syntactically correct.

Run-Time Environment

- objectives: 1. Predict what will happen when a program is running.
2. Describe how the program runs.
3. Differentiate between static and dynamic environments.

Program Execution

- objectives: 1. Follow program execution by hand in all its steps.

Run-Time Storage Management

- objectives: 1. Define static (as in FORTRAN and COBOL) and dynamic (as in Pascal and PL/1) storage management.
2. Draw the main parts of the run-time stack and show how it changes as the program executes.
3. Describe how recursion affects storage management.

Parameter Passing Mechanisms

- objectives: 1. Define the main parameter passing mechanisms (call-by-value, call-by-reference, call-by-name).
2. Show how parameters affect local and global entities at run-time (eg. side-effects, using the same name in different contexts/places)

5. LOGIC DESIGN

AIMS: 1. Understanding of how the components of a computer inter-relate.

Boolean Algebra

- objectives: 1. Define and evaluate boolean expressions.
2. Describe the relationship between boolean algebras and computer circuits.

Mapping (Simplifying Boolean Circuits)

- objectives: 1. State and describe DeMorgan's theorems.
2. Draw 2-6 variable maps.
3. Write sum-of-products boolean expression for various truth-tables)

Digital Circuits

- objectives: 1. Draw truth tables for boolean functions
2. Define gate.
3. Draw the corresponding gates for given boolean expressions.
4. Define switching circuits.
5. Draw and/or build specified circuits.
6. Define masking.

Combinational Circuits

- objectives: 1. Differentiate between sequential and combinational circuits.
2. Describe, draw some examples of combinational circuits (eg. AND, OR, NAND)

Sequential Circuits

- objectives: 1. Describe, draw, and build the following circuits: flip-flops, counters, shift-registers)
 2. List several applications.
 3. Define state-reduction.
 4. Show how a simple circuit might be reduced.

Data Representation

- objectives: 1. Draw and explain how data is represented at the bit level.

Composition

- objectives: 1. Define bit, byte, and word, and describe how they are related.

Codes

- objectives: 1. Define the term binary code.
 2. Describe how bits, bytes and words relate to characters and binary codes.

Error Detection and Correction

- objectives: 1. Compare 2 methods of error detection and correction.

Numbers

- objectives: 1. Describe and manipulate the most common number bases (binary, decimal, octal, hexadecimal, BCD)

Types

- objectives: 1. Name the commonly used number bases.
 2. State the algorithms for conversion.

Representation

- objectives: 1. Describe several common number formats (eg. binary, BCD)
 2. Describe how overflows occur and how they can be detected.

Digital Arithmetic

- objectives: 1. Draw and explain how digital arithmetic is accomplished.

Addition

- objectives: Diagram the function of an adder (half, full, serial, parallel)

Subtraction and Negation

- objectives: 1. Convert ordinary negatives to 1's and 2's compliment and signed magnitude.
 2. Describe subtraction algorithms

Multiplication and Division

- objectives: 1. Describe common multiply and divide algorithms.
 2. Discuss methods for speeding up these algorithms.

Floating Point

- objectives: 1. Draw representation of a floating point number on a familiar system.
 2. Contrast floating point arithmetic with fixed point
 3. Define normalization.

Logic Control

- objectives: 1. Define hard-wired control.

2. Define microprogrammed control.
3. Discuss how the processor unit is controlled.

Multiplexors

objectives: 1. Describe their purpose and function.

D/A = A/D Conversion

objectives: 1. Describe, in general terms their purpose and function.

Digital Integrated Circuits

objectives: 1. Some understanding of how the circuits are built and operate.
 2. Define diodes and transistors and describe their purpose.
 3. Describe the common types of DIC's, their differences and uses (eg. MOS, CMOS, TTL, IIL, ECL, etc.)

Processor Logic Design

objectives: 1. Describe the main steps involved in designing a processor; including: organization, logic circuit design, arithmetic circuit design, ALU design, status register, shifter, accumulator.

Computer Design

objectives: 1. Describe the main steps in designing a computer, including: system configuration, instructions and execution, timing and control, register design, control design, 'the console'.

6. ARCHITECTURE

AIMS: 1. Understanding of how a machine works, how instructions are carried out, how data is managed.
 2. Understanding of what is happening in the machine when a program is compiled and run.
 3. Knowledge of how to manipulate data at the bit level.
 4. Understanding of how representation of data and word-size affect architecture.
 5. Provide a concrete example of a machine architecture and how to write programs which use that architecture.

Von Neumann Machine

objectives: 1. Describe and draw the operation of the Von Neumann machine.
 2. Discuss various methods of implementation (eg. stack machines, register machines) and their applications.

Hardware Systems Organization

objectives: 1. State the major characteristics (functions) of and relationships between I/O devices, processors, control units, main and auxiliary storage devices.
 2. Define bootstrapping and describe the process in general terms.

Memory

objectives: 1. Describe the role that memory plays in the hardware system.

Types

- objectives: 1. Describe RAM, ROM.
- 2. List the different types of memory (eg. virtual, cache, interleaved, associative) and contrast their features.

Access and Control

- objectives: 1. Describe how memory (RAM) is controlled and accessed.

Paging and Segmentation

- objectives: 1. Define paging and segmentation.
- 2. Describe advantages and disadvantages of each.
- 3. Define trashing.

Storage Allocation

- objectives: 1. Define the need for storage allocation.
- 2. Describe two techniques for performing the task.
- 3. Define spooling.

Instructions

- objectives: 1. Describe the different types of instructions and explain how they work.

Principle Types

- objectives: 1. List principal types of instructions (eg. register, logical, arithmetic, jump, etc.) and describe how they differ.

Format

- objectives: 1. Recognize how instructions are defined in terms of bit sequences (eg. codes — size, number)
- 2. Illustrate how registers and addresses may be specified.

Execution

- objectives: 1. Describe in general terms, how the different types of instruction fetch, decode, etc.)
- 2. Define the concept of system state.
- 3. Describe how the system state can change and how it can be saved and restored.

Implementation

- objectives: 1. Discuss various methods of implementing instructions (eg. hardware, microcode).

Error Conditions

- objectives: 1. Describe how systems signal and deal with error conditions (eg. error flags for overflow)
- 2. Describe how error conditions can be tested for, what they mean, what can be done about them.

Addressing

- objectives: 1. Describe the various addressing techniques, their differences, similarities and uses.
- 2. Describe how the organization of the machine facilitates different modes of addressing.

Symbolic Coding

- objectives: 1. Describe how symbols are defined and assembled.

2. State how symbols affect addressing.

Literals

- objectives: 1. Describe the definition of literals.
2. Discuss how and when they are used.

Mnemonic Op Codes

- objectives: 1. Using a known assembler, show how the mnemonic op codes correspond to the actual op codes.

Symbolic Address

- objectives: 1. Show how symbolic addresses (including address expressions and labels) are defined and evaluated.
2. Define relocation and state which symbols are relocatable and which are absolute.

Pseudo-Ops

- objectives: 1. Discuss their uses.
2. Describe the difference between pseudo-ops and other instructions.

Assembler Language

- objectives: 1. Write assembler language programs and follow their execution.
2. Illustrate how a known architecture can be used to advantage.

Subroutines

- objectives: 1. Describe the function and use of subroutines and how jumps occur.
2. Discuss parameter passing mechanisms.

Programming

- objectives: 1. Write assembler programs.
2. Suggest how the assembler might look for constructs in higher level languages.
3. Implement high level language constructs in assembler.

Assembly, Scanning

- objectives: 1. Follow the scanning process and describe what the scanner is doing.
2. Assemble and disassemble programs by hand.
3. Describe the operation of a two-pass assembler.
4. Describe how to resolve forward references.

Symbol Tables

- objectives: 1. Describe the purpose of symbol tables.
2. Follow the assembly of a program and draw the symbol table.

Program Segmentation and Linkage

- objectives: 1. Know how programs are prepared for execution.

Loading and Linking

- objectives: 1. Define the terms.
2. State the need.
3. Describe how programs are loaded and linked where applicable.

Separate Compilation, Resolving References

- objectives: 1. Explain how routines may be separately compiled.

2. Explain how references are resolved for separately compiled code.

Subroutines, Functions, Coroutines, Re-entrant Subprograms

- objectives: 1. Distinguish the different types of routines
2. Discuss how each type is assembled and run.

Parameter Passing and Binding

- objectives: 1. Define binding.
2. Explain how parameters are passed and bound.

Relocation

- objectives: 1. Define relocation.
2. Explain how relocation occurs and what is involved.

Overlays

- objectives: 1. Define overlay.
2. Discuss how they work.

Macros

- objectives: 1. Describe the concept of a macro.
2. Discuss their uses.

Definition, Call and Expansion

- objectives: 1. Write macros for given purposes.
2. Predict how they will be expanded.

Assembly Time Computation

- objectives: 1. Differentiate between those expressions that are evaluated at assembly time and those that are evaluated at run time.

Conditional Assembly

- objectives: 1. Define conditional assembly.
2. List several situations where it might be useful.

Parameter Handling

- objectives: 1. Describe how parameters are handled in macros.
2. Contrast parameter handling in macros with parameter handling in other routines.

Hardware Control

- objectives: 1. Compare and contrast synchronous and asynchronous control
2. Describe several modes of communication between devices
3. Discuss several issues related to reliability and how they are dealt with

I/O Operation and Devices

- objectives: 1. Describe how data is moved from one location to another (eg. data transfer)
2. Define data bus and describe its purpose
3. Describe handshaking.

Interrupts

- objectives: 1. Describe what happens when an interrupt occurs
2. Define the term interrupt
3. List the conditions under which an interrupt may occur
4. Write an interrupt routine.

Microprogramming

- objectives: 1. Define the concept of microprogramming
 2. Contrast microprogramming with assembler programming
 3. Describe the kinds of operations accomplished by microprograms
 4. Write a microprogram routine and describe how it works.

Microprocessors

- objectives: 1. Distinguish microprocessors from other types of processors
 2. Discuss how microprocessors may be used in computer systems and in their applications (eg. cars, appliances).

Multiprocessors

- objectives: 1. Define the term
 2. Describe how they differ from single processors in terms of control and programming.

7. DISCRETE MATHEMATICS

- AIMS: 1. Discussion of required background in discrete mathematics for further study in the theory of computation
 2. Form an axiomatic foundation for elementary concepts in discrete mathematics.

Matrices and Vectors

- objectives: 1. Define the general properties
 2. Perform addition and multiplication
 3. Calculate the transpose
 4. Perform inversion
 5. Calculate determinants.

Set Theory

- objectives: 1. Define the attributes and properties
 2. Describe and draw Venn diagrams
 3. Perform set operations
 4. Define classes of sets
 5. Define the term power set
 6. Define maps, regions
 7. Contrast, prove various relations, functions.

Graphs

- objectives: 1. Define attributes and properties
 2. List applications of graphs
 3. Define classes of graphs (eg. trees, directed graphs)
 4. Discuss important graph theoretical problems (eg. spanning trees, shortest path, max flow min out)
 5. Describe combinatorial and counting problems (eg. counting rooted trees)

Abstract Algebra

- objectives: 1. Define 'semigroup'
 2. Define attributes and properties of groups, semigroups, and fields
 3. Define free monoid.

4. Discuss applications of groups, semigroups, and fields.

Proofs

- objectives: 1. Solve problems by strict reasoning using a mathematical (analytical) approach
2. Construct rigorous proofs
 3. Explain the principle of induction
 4. Perform induction proofs
 5. Recognize and describe applications for other proofs (eg. contradiction, constructive)

8. THEORY OF COMPUTATION

AIMS: 1. Provide a general introduction to theory and problems in Computer Science

Finite Automaton

- objectives: 1. Define regular expressions and how they can be recognized
2. Draw finite state machines for given grammars
 3. Describe how recognizers work
 4. Define equivalence relations
 5. Present canonical forms.

Formal Languages and Grammars

- objectives: 1. Present definitions, formal notations
2. Define alphabets
 3. Describe various types of grammars
 4. Define the empty sentence
 5. Define 'recursiveness' in relation to grammars
 6. Construct and describe derivation trees.

Context-Free Grammars

- objectives: 1. Define and diagram context-free grammars
2. describe and identify context-free grammars
 3. Formulate derivation trees
 4. Define closure
 5. Describe push-down automata.

Context-Sensitive Grammars

- objectives: 1. Present definitions
2. Describe difficulties in dealing with them.

Turing Machine

- AIMS:** 1. General understanding of the theoretical limitations on the computer's capabilities
- objectives: 1. Define the concept of a turing machine
2. Present definitions and notation

3. Define related concepts (eg. primitive recursion functions, recursively enumerable sets, Church's thesis)
4. Identify the halting problem

Computability

objectives: 1. Describe the problem.

Analysis of Algorithms

- objectives: 1. Define the concepts of space and time efficiency
2. Differentiate between deterministic and non-deterministic automata
 3. Define the concept of np-completeness
 4. Illustrate the equivalence of some important np-complete problems
 5. Discuss several algorithms for manipulating graphs (eg. minimal spanning algorithms, depth-first search, breadth-first search)

Information Theory

- objectives: 1. Define information theory
2. Relate the work of Shannon and discuss his role in information theory
 3. Define 'measure of information'
 4. Define entropy and variety
 5. Describe at least two types of codes (ASCII, Huffman)
 6. Discuss error detection and correction techniques of these codes
 7. Define modularity.

9. DATA STRUCTURES

- AIMS: 1. Familiarity with the use of common data structures
2. Implementation of common data structures in several high-level programming languages

Record Types

- objectives: 1. Describe their representation
2. Write programs to manipulate records
 3. Discuss assignment of different parts of the record (as is 'move corresponding' in COBOL and 'by name' in PL/1)

Pointers

- objectives: 1. Define pointers in terms of programming languages
2. Show how pointers are used to aid in the manipulation of other data structure.

Sets

- objectives: 1. Describe their representation
2. Write programs to manipulate sets
 3. Recognize appropriate applications.

Lists

- objectives: 1. Distinguish different types of lists, their uses and applications.

Arrays

- objectives: 1. Describe their representation (fixed and variable length, single and multidimensional)
 2. Calculate subscripts
 3. Write programs to manipulate arrays
 4. Describe applications of arrays (eg. sparse, triangular matrices)

Strings

- objectives: 1. Diagram the representation of strings — both fixed and varying length
 2. Describe the common string operations
 3. Write programs that manipulate strings.

Stacks and Queues

- objectives: 1. Draw stacks and queues
 2. Describe how they would be implemented using other data structures
 3. Write routines to manipulate stacks and queues.

Linked Lists

- objectives: 1. Draw linked lists (eg. singly-linked, doubly-linked, circularly-linked)
 2. Demonstrate insertion, deletion, searching, and traversal of the various linked lists
 3. Show how the various linked lists can be implemented.

Trees

- objectives: 1. Define the terminology
 2. Distinguish the common types of trees (eg. binary, n-ary)
 3. Show how trees can be implemented
 4. Demonstrate the creation of trees and tree-nodes
 5. Demonstrate, diagrammatically, and with programs, how trees can be traversed (in-order, pre-order, post-order)
 6. Demonstrate algorithms for searching and sorting trees
 7. Define treading and discuss its effects on insertion, deletion, traversal
 8. List several applications of trees.

Sorting and Searching

- objectives: 1. Describe and discuss various algorithms for sorting and searching information and how they are related
 2. Define hashing
 3. Discuss several hashing techniques and identify their role in sorting and searching algorithms.

10. STORAGE MANAGEMENT

- AIMS: 1. General understanding of how storage on a computer system is managed and how different types of files are manipulated

Memory Storage

- objectives: 1. Distinguish dynamic from static storage management
 2. Describe what happens when variables are allocated and deallocated
 3. Diagram storage management using heaps and stacks
 4. Define garbage collection
 5. Identify the need and describe, in general terms, how it works.

Files

- objectives: 1. Define the terminology (eg. record, file, blocking, etc.)
 2. Describe the types and characteristics (eg. capabilities, speed) of the common mass storage media (eg. disk, tape)
 3. Describe the main type of files (eg. sequential access, random access) and their characteristics
 4. Demonstrate diagrammatically how these files are manipulated
 5. Describe how files are written and read and illustrate the programming language constructs used
 6. Discuss sort/merge algorithms.

11. OTHER MAJOR TOPICS

AIMS: 1. Some understanding of the main areas in Computer Science (i.e. familiarity with the terms and some idea of what they are about)

Numerical Methods

Word Processing

Data Processing

Random Numbers

Real-time Programming

Translators

Graphics

Simulation

Databases

Distributed Systems

Networks

Communications

Artificial Intelligence

Robotics and Cybernetics

Pattern Recognition

12. MISCELLANEOUS

AIMS: 1. General introduction to other issues relating to Computer Science

History of Computers

- objectives: 1. Trace the major steps in the development of computers since ancient times
2. Discuss some of the motivations and problems.

Social Issues in Computer Science

- objectives: 1. Discuss the impact that computers have had on society
2. Discuss how computers have changed various aspects of our lives
3. Discuss privacy, regulation, computer abuse and crime
4. Discuss trans-border data flow.

Computers and the Law

- objectives: 1. Discuss current issues regarding copyrights, patents, proprietary rights, trade secrets
2. Discuss writing computer contracts for software and hardware maintenance
3. Describe some of the laws currently in place regarding information held in the computer (governmental regulation and taxation)

The Computer Industry

- objectives: 1. Discuss current markets, supplies
2. Discuss standards for software, conduct.