JAMES COOK UNIVERSITY
OF
NORTH QUEENSLAND

Teaching Computer Science as the Science of Information

Gopal K. Gupta

TR 96/10

DEPARTMENT OF COMPUTER SCIENCE

TOWNSVILLE
QUEENSLAND 4811
AUSTRALIA

Title	Teaching Computer Science as the Science of Information
Primary Author(s)	Gopal K. Gupta
Contact Information	Gopal K. Gupta Department of Computer Science James Cook University Townsville, QLD 4811 AUSTRALIA `gopal@cs.jcu.edu.au`
Date	July 22, 1996

Teaching Computer Science as the Science of Information

Gopal K. Gupta
Department of Computer Science
James Cook University
Townsville, Qld 4811
Australia

Abstract

During the last thirty years a number of model curricula for computer science have been developed but computer science curriculum continues to be a topic of intense discussion (see, for example, the debate in the December 1989 issue of the Communications of the ACM). The introductory computer science teaching continues to be problematic with many departments reporting high drop-out and high failure rates in the introductory courses. Many students feel great deal of learning frustration and the introductory courses have been criticized for, among other things, too much material, lack of insight-building, and programming before reasoning.

We believe that the conventional curricula and the solutions proposed to overcome the problems of the curricula share a similar approach to introductory teaching which essentially involves teaching procedural programming using an apprenticeship approach. We believe that many of the problems with conventional introductory curricula arise as a result of following this basic approach and it is this approach that needs to change. We suggest an alternate approach to teaching computer science that is based on information, its processing, presentation and communication as the focus of computer science rather than procedural programming. Programming continues to be an important part of the proposed curriculum but it does not occupy the central place that it does in many current curricula. The curriculum includes a number of software engineering group projects and the programming and software development learning takes place in these group projects. A detailed introductory curriculum is presented and a framework for designing the whole computer science curriculum is discussed.

1. Introduction

Computer Science curriculum has been a topic of intense discussion since the birth of the discipline in the early 1960's (see, for example, the debate in Denning, 1989). A number of model curricula, including Curriculum 68, Curriculum 78 and, Curriculum 1991, have been developed by ACM and have been widely used as basis for curriculum design. A number of other computing societies have also had their own recommendations. Tucker and Wegner (1994) review the evolution of the ACM curricula and discuss some alternatives.

Curriculum 68 proposed a core curriculum of four basic courses: algorithms and programming, computer and system structure, discrete structures, and numerical calculus, followed by four intermediate courses: data structures, programming languages, computer organisation, and system programming. Curriculum 78 placed emphasis on algorithms, programming, data structures, and hardware and proposed that every computer science graduate should know: how to write programs, measure the efficiency of programs, know what problems are amenable to computer solution, understand individual and team problem solving, understand computer architectures, and be prepared to pursue in-depth training or graduate study in computer science. In 1985, the ACM appointed a task force on the core of computer science. The report of this task force (Denning *et al*, 1989) defined computer science as *the systematic study of algorithmic processes that describe and transform information; their theory, analysis, design, efficiency, implementation, and application*. Nine areas of the discipline were identified: algorithms and data structures, programming languages, architecture, numerical and symbolic computation, operating systems, software methodology and engineering, database and information retrieval, artificial intelligence and robotics, human-computer communication. Based on this report, Curriculum 91 was developed that encourages innovation in curriculum design and recommends a number of core courses (about 270 lectures, rather too large a core for a three-year degree program) based on the nine areas listed above. The core is dominated by algorithms and data structures (47 lectures), architecture (59), programming languages (46) and software methodology and engineering (44) leaving only 75 lectures for the remaining five areas.

Given the above recommendations, the introductory computer sciences courses continue to teach problem solving, algorithm design, procedural programming, debugging and testing, but we show in the next section that such teaching continues to be problematic with many departments reporting high drop-out and high failure rates in such courses. Many students feel great deal of learning frustration and the introductory courses have been criticized for, among other things, too much material, lack of insight-building, and programming before reasoning. It is therefore essential that alternative approaches to teaching computer science be explored. We explore one alternate approach in this paper.

The aim of this paper is to show that a viable alternate approach to teaching computer science is possible and is worth considering. In the next section we first discuss why the present approach has not been successful and the problems that have been identified and some solutions that have been proposed. This is followed, in Section 3, by the proposed new approach for introductory computer science. In Section 4 we present a basic structure for building a whole new computer science curriculum. Section 5 concludes the paper.

2. Introductory Computer Science Courses

Designing introductory computer science courses is particularly difficult since introductory courses often try to meet a number of objectives which are not always compatible. Also, the students attending introductory computer science courses often have very diverse backgrounds in mathematics and computing and the curriculum must try to deal with the diversity. It is possible to have courses that either emphasise programming but ignore breadth of coverage or have a breadth-first course that gives up some of the programming emphasis. Often though, the first one or two courses are dominated by programming and Tucker and Wegner (1994) note that Brown University CS1 course requires students to write several thousand lines of code in the first semester.

Most introductory computer science courses have objectives similar to those listed by Koffman, Miller and Wardle (1984) who present a model curriculum for an introductory course CS1. These objectives are:

- to introduce a disciplined approach to problem-solving methods and algorithm development
- to introduce procedural and data abstraction
- to teach program design, coding, debugging, testing and documentation using good programming style
- to teach a block-structured high-level programming language
- to provide a familiarity with the evolution of computer hardware and software technology
- to provide a foundation for further studies in computer science

Koffman *et al* (1984) present course details including information on the programming language to be used and how the course should be administered and delivered. They note that *programming assignments comprise a significant part of the student workload*.

In spite of the model curricula and some experimentation with them, the most intense discussions in computer science curriculum design continue to be those related to introductory courses (for recent examples, refer to Tucker and Wegner, 1994, Scragg, Baldwin and Kooment, 1994, Doran and Langan, 1995). For example, the papers by Scragg *et al* and Doran and Langan present what in their view are symptoms of a number of problems with introductory computer science courses. A list of problems that are responsible for these symptoms, in the view of these authors, are then identified and solutions proposed. Shaw(1990) also lists a set of flaws in introductory courses.

Some of the symptoms identified by the authors of the papers cited above are high drop-out rates in the undergraduate programs, in particular in the introductory courses, complaints by the employers that the graduates are not able to apply what they have learned and high drop-out rates from computer science PhD programs (which presumably reflects poor preparation). Although these symptoms are by no means found in all computer science programs, they are common enough to be familiar to most computer science academics. Bagert, Marcy and Calloni (1995) give an extreme example of drop-out and failure rates in which a class of 216 students in first year was reduced to only five graduates four years later.

Some of the problems that have been identified are:

(1) great deal of learning frustration

(2) poorly defined exit behaviour

(3) too much material

(4) inappropriate emphasis on design

(5) lack of insight-building

(6) programming from scratch

(7) equating program text with software

(8) programming before reasoning

(9) throwaway exercises

Yet other problems have been noted. For example, it has been noted that a computer science curriculum is often an extended list of topics which attempts to survey the field of information technology. Given the wide scope of the field, a survey of the whole field is generally not possible and, given the dynamic nature of the field, a survey may not even be desirable. Although many curricula have a large core, a student majoring in computer science may graduate without having any conceptual understanding of how even the very basic software (e.g. a word processor, a spreadsheet or a simple DBMS) is designed or works. In addition, the current computer science programs do not cater for the variety of needs of the computing industry of today and tomorrow. Given that the courses for major and non-majors are often different, the present computer science curriculum does not appear to encourage cross-disciplinary education which the National Research Council committee (Hartmanis and Lin, 1992) believed to be so important.

A problem that has not been identified in this context is that the present computer science introductory courses appear to be *not attractive to many female students*. Klawe and Leveson (1995) discuss some of the reasons for low percentage of female students in most computer science classes and note the role of parents and teachers in shaping attitudes of girls in schools. Girls also appear to have less access to computers at home compared to boys. Whatever the reasons, anecdotal evidence suggests that many female students do not like the emphasis on programming that is common in introductory courses.

A number of other issues about introductory computer science have been raised in the literature. These include the following:

(a) what programming language should be used in the introductory course?

(b) how should problem solving be taught?

(c) how should program design be taught?

(d) when should object-oriented concepts be introduced?

(e) what is the role of formal methods in introductory computer science?

(f) does computer science curriculum needs more or less mathematics?

Most of these discussions share the basic assumption that an introductory course should have objectives similar to those listed by Koffman *et al* (1984).

A number of solutions have been proposed to resolve the problems listed above. For example, Shaw(1990) suggests that an introductory course should include a study of good examples of software systems, learn more facts, modify and combine programs as well as creating them, incorporate reference material as it becomes available, and present theory and models in the context of practice. The ACM task force (Denning *et al*, 1989) claims that fundamentals of the discipline are contained in three basic processes - theory, abstraction and design in the nine areas listed earlier. Based on this report, Baldwin *et al* (1994) and Scragg *et al* (1994) state that the central mission of their introductory sequence is to teach design, theory and empirical analysis. Doran and Langan (1995) take a very different approach. They refer to the six levels of learning in educational process viz. knowledge, comprehension, application, analysis, synthesis and judgement and comment that the introductory courses should try to primarily teach the first three levels since the other three, analysis, synthesis and judgement, require a good mastery of the first three and maturity gained by extended usage.

Although the above solutions are likely to improve introductory courses they are unlikely to overcome the problems that arise because of the focus on algorithmic computation using an apprenticeship approach. Astrachan and Reed (1995) in fact explicitly recommend an apprenticeship approach in which students read, study and extend programs written by experienced and expert programmers while others recommend group learning, e.g. Sabin and Sabin (1994).

Many conventional introductory courses expect students to complete programming assignments without a great deal of assistance. Basically, given the examples in the class, the students are expected to obtain the solutions to the assignments by *discovery* while they are often still learning the syntax of a programming language. Pennington and Grabowski (1990) discuss the tasks involved in programming and they note that "*..programmer must comprehend the problem to be solved by the program, design an algorithm to solve the problem, code the algorithm into a conventional programming language, test the program and make modifications in the program...In sum, programming is a complex cognitive and social task composed of a variety of interacting subtasks and involving several kinds of specialized knowledge..*". Programming therefore is a creative activity which involves synthesis of a variety of knowledge and, as noted earlier, is therefore a higher level learning process that requires mastery of at least knowledge, comprehension and application. It may be that requiring synthesis early in an introductory course is bound to create problems if the student has not acquired the relevant knowledge, comprehension and application prior to starting the course.

Although the above approach of teaching introductory courses for the computer science majors is common, some educators recommend a similar approach for non-majors. For example, Biermann (1994) recommends that a significant part of a course for non-majors be programming. The author claims that one of the reasons for teaching programming in an introductory course is that *it catches the attention of students....Most students enjoy the [programming] experience and want to show off their programs*! The author notes other reasons for teaching programming and notes that programming conveys a kind of intuition about computers and programming experience is excellent for teaching notation. This is hardly a convincing argument in our view.

The conventional approach we believe has several significant disadvantages. The apprenticeship approach appears suitable only for learning trade or technical skills and is perhaps not suitable for learning conceptual material. Also, apprenticeship approach is often ineffective due to a lack of staff resources and results in average students constantly battling with programming assignments and thus having no time to consider the conceptual basis of the discipline. Due to heavy assignments workload many such students drop out or switch to other disciplines after (or before!) completing the first course in computer science. Also, the introductory curriculum itself often is not particularly suitable since it conveys to the students a very limited view of computer science. The students are given the impression that a computing professional is constantly been given problems to solve which he/she must find solutions to and code the solutions and debug and document that code. This is far from the true nature of work of a computing professional.

Given the conventional curriculum, it is no wonder that many students remark that computer science is just programming; that is what we tell the students in our introductory courses as noted by Denning *et al* (1989):

> The view that "computer science equals programming" is especially strong in most of our current curricula: the introductory course is programming, the technology is in our core courses, and the science in our electives. This view blocks progress in reorganizing the curriculum and turns away the best students, who want a greater challenge. .....The emphasis on programming arises from our long-standing belief that programming languages are excellent vehicles for gaining access to the rest of the field, a belief that limits our ability to speak about the discipline in terms that reveal its full breadth and richness. .... *Clearly programming is part of the standard practices of the discipline and every computing major should achieve competence in it. This does not, however, imply that the curriculum should be based on programming or that the introductory courses should be programming courses.* [emphasis added]

Another significant disadvantage of the conventional approach is that the introductory curriculum is too dependent on changes in technology which an introductory course ought not to be. New programming languages and new programming paradigms often need to be reflected in the introductory courses and the curriculum discussions in computer science departments are often dominated by whether the programming language being used needs to be changed to the latest one to the neglect of more important issues. Such language discussions have been known to turn into religious wars that can do serious damage to the fabric of a department. Programming of course not only plays a central role in introductory courses, it tends to play a central role in most subsequent courses since the attitude that one can only learn by implementing pervades the whole conventional computer science curriculum. Given this approach, a change in programming language used in the introductory course can bring havoc to the whole curriculum which can consume enormous amount of invaluable staff resources in revising the course materials.

To summarise, we believe the current introductory computer science courses have the following problems:

(1) the curriculum focuses on programming and algorithmic computation and does not present a broad picture of computer science

(2) the curriculum requires too much programming which is often intimidating for at least a significant minority of students, perhaps more so with the female students

(3) the curriculum uses an apprenticeship approach which is unsuitable for some types of learning and the approach is often very time consuming for most students given the lack of staff resources; the high workload often results in students either dropping out of the computer science course or neglecting their other studies

(4) the pass rates in courses based on such curriculum are often low and the drop-out rates high; retention of less than 50% after first year courses appears common but in our view unacceptable

(5) although the programming assignments can be time-consuming, the curriculum conveys few intellectual challenges to some of the brightest students

3. A New Approach to Teaching Computer Science

We believe a different approach to teaching computer science is needed not only because the present approach suffers from significant difficulties but also because computing and the computing industry have changed dramatically over the last four decades. In the early days the primary concern was to keep the hardware running and its efficient use. With more reliable hardware, the concern shifted to system software and then to application software. The concern is now shifting to information storage, retrieval, display and presentation. As a result, the role of programming and growth in jobs for programmers have diminished considerably in the industry (Keaton and Hamilton, 1996) and instead there are significantly more opportunities for people with a variety of other skills (e.g. networking, Web applications). We believe a new curriculum should meet a variety of needs of the industry, not be technology driven, provide a good introduction to computer science, and be realistic for the educational environment. To be realistic, for example in Australia, the curriculum must take into account the fact that a large number of students now enter universities without adequate preparation in mathematics. Furthermore, the curriculum must take into account that not all students studying computer science are going to be very proficient in programming and software development and some may not be capable of or interested in becoming software developers although still interested in developing some computing knowledge and expertise. Furthermore, many degree programs, for example those in Australia, are of only three-year duration and therefore the course time is a very limited resource which must be used wisely. This clearly suggests that proposals like those of Dijkstra (Denning, 1989) in which he suggests that each program must be accompanied by a formal proof that it meets the formal specifications and those of Parnas (1990) in which he proposes a computing program that is a five-year engineering degree (and does not even provide an introduction to data base management) are not worth considering since they propose solutions that in our view are unrealistic.

Computer systems consist essentially of hardware, software and the information that they process. Taking a rather simple view, one could say that these three components are the centre of attention in the three computing disciplines viz. computer systems engineering, computer science, and information systems. The information systems view of information is however very limited since that discipline is primarily concerned with role of information in decision-making. Information has many dimensions and in fact meets many needs in addition to that of decision-making and all computing essentially deals with information. We propose that a very much broader view of information be the basis of computer science curriculum and we agree with Denning (1995) that a computer science program should be taught as the science of information. Computer science should therefore have the concept of information, its manipulation, communication and display as its focus rather than algorithmic computation. As Denning notes, information is a powerful metaphor, often compared with fluid that can flow, have a source, and be extracted, transformed, acquired and contained; data, symbols, signals and messages are carriers of the fluid of information. Just as phenomena surrounding fluids are worthy of scientific study, so are phenomena surrounding information, only more so since information is now so much more widely used.

If information, its manipulation, communication, and display are to be the central concern of computer science, the introductory computer science course as well as the courses that follow will be significantly different than what we teach today. As an example, we propose that an introductory course consist of the following topics; the details are presented in the next section:

(1) the concept of information, its many forms, its storage and retrieval; its value, the need for many different types of information manipulation.

(2) computers as (simple) machines that manipulate information; a simple introduction to computer organisation.

(3) the need to input information to computers and to output information from computers so that input information may be manipulated and manipulated information displayed; the need for many different types of input and output.

(4) the need to store information in computers; techniques for storing and retrieving information.

(5) the concept of ownership, availability and fair use of information; issues of copyright, personal privacy, information rich and information poor, ethics.

(6) the need to communicate information from an input device to a computer and from a computer to an output device; communication of information from one computer to another.

(7) the need to manipulate information (mathematical computations, symbolic computations, intelligence); techniques for simple information manipulation, complex information manipulation.

(8) instructing the machine to manipulate information; introduction to one procedural language; examples of algorithms and their implementations using the language, debugging, testing, and documentation.

3.1. A Detailed Curricula for CS1

We now present details of the lectures and the laboratory classes of the introductory curriculum:

(1) **The concept of information and its many forms:**

What is information?: Its definition and characteristics.

Temporal and non-temporal information.

Non-temporal information: Text; linear and non-linear text; Linear Text: text without form and text with form; text without form, sequence of characters, how many different characters, ASCII, ISO character sets, character sets for LOTE;

Text with form, need to store content and form both, storing form by using mark-up languages (troff, latex, SGML, ODA), storing form by specifying form in a WYSIWIG editor;

Presentation, fonts, device independent fonts, storage and printing of fonts, geometric descriptions of fonts, kerning, PostScript.

Non-linear form of text: hypertext, representation of hypertext, Web.

Operations on text: retrieval, character and string operations, editing, formatting, pattern-matching and searching, spell checking, style checking, compression, encryption.

Images: Image as a two-dimensional array of pixels, monochrome, grey and colour images; colour models, representation of images. Operations on images: editing, point operations, filtering, compositing, geometric transformations, conversions, compression.

Graphics: Difference between graphics and image data; representation of graphics data, geometric modelling (GKS, PHIGS, etc), solid models, other models. Operations on graphics data: editing, shading, mapping, lighting, viewing, rendering.

Temporal Data - Video: Video as sequence of images or frames, analogue video and digital video, analogue representation and major formats (NTSC, PAL, SECAM, etc), video storage, digital representation, data rates, digital video storage. Operations on video: video sources and sinks, video mixer, retrieval, editing, compression of digital video, MPEG, JPEG, etc.

Temporal Data - *Audio*: Digital and analogue audio representation, speech, encoding, audio formats (CD, DAT, etc). Operations: storage, retrieval, editing and filtering.

Temporal Data - *Music*: Difference between music and audio; representing music (MIDI, SMDL). Operations: playback, synthesis, editing, composition.

Animation - Sequence of synthetic image frames, animation vs video, animation models. Operations: motion and parameter control, rendering.

Information transformation - Transforming text to audio, speech to text, music to audio, animation to video, etc. Refer to Gibbs and Tsichritzis (1994) page 77.

Optional: What is knowledge? representing knowledge e.g. plans, games, rules, etc.; natural language representation and understanding.

Laboratory: Students use softwares that processes textual information e.g. mark-up languages, WYSIWIG editors, hypertext using the Web and carry out the operations specified including compression and encryption. Importance of information presentation.

Students handle a variety of images and use software that carries out operations on images. Graphics information. Video, audio, speech, music and animation information and use of softwares for variety of operations on these types of information.

(2) **Information storage and retrieval, value of information, need for many different types of manipulation, etc** .

Manual storage and retrieval of information in society; organisation of large amounts of information on paper; books, chapters, table of contents, index, telephone directory, children's story books that allow the reader some options on how the story develops, organisation of books in library, files and filing cabinets in a office, inventories, archiving, introduction of terms like sequential, index sequential, trees; analogies between manual information handling and information handling by computers.

Information as a commodity and its unique properties, needs of graphical display of information, computing aggregate values, and other manipulations.

Laboratory: A study of manual information storage and retrieval. Discussion of different possible techniques, introduction to some computing terminology. Discussion of information as commodity and its value, examples of valuable and useless information. Importance of information presentation.

(3) **Computers as (simple) machines that manipulate information; simple introduction to computer organisation.**

Conceptual model of a computer, computer examples: computer games, ATM, a computer in a washing machine or a car, large mainframes, desktop, laptop and mobile computers. Concept of instructing computers; analogies between instructing a computer and instructing a human worker.

Laboratory: Explore students' prior experiences with computers, their understanding of what is inside a computer, show inside of a computer to students, discuss essential components and logical organisation.

(4) **The need to input information to computers and to output information from computers so that information may be manipulated and manipulated information may be displayed; The need for many different types of inputs and outputs.**

Input to computers via keyboard, scanners, cameras, speech, instruments, touch screens, pen, mouse, joystick, etc., conceptual understanding of how these devices work, importance of of devices like scanners and fax and their working; importance of user-friendly information input.

Output from computers via monitors, printers of different types, speech, music, graphical output, control signals, etc; conceptual understanding of how these devices work, importance of user-friendly information output.

Laboratory: Use a variety of input and output devices, understanding of how these devices work, importance of user-friendly interaction with computers.

(5) **The need to store information in computers, techniques for storing and retrieving information.**

Introduction to storage devices: core memory, disks, CD-ROMS, tapes etc, introduction to data structures (sequential, indexed sequential, trees, hashing), concept of relational database and a simple retrieval language; analogies between manual and computer storage and retrieval of information.

Laboratory: Recall techniques of manual storage and retrieval of information; present examples where data structure is important; simple relational database representation, simple retrieval language, use of a PC database system.

(6) **The concept of ownership, availability and fair use of information; issues of copyright, personal privacy, information rich and information poor, ethics.**

Who owns information? access to information held by public and private organisations, intellectual property rights, concept of information privacy, information privacy principles, information society, concept of information rich and poor societies, ethics for information processing professionals.

Laboratory: Recall value of information discussion, difficulties in obtaining relevant information, role of technologies like the Web in making information accessible, discussion of ownership of information and intellectual property, IP rights on the Web, cases of violation of privacy using database systems, credit data, medical data,etc, ethical behaviour by computing professionals, code of ethics of medical and legal professions.

(7) **The need to communicate information from an input device to a computer and from a computer to an output device; also sending information from one computer to another.**

Computer communications, email, internet, www, networks, mobile computing, conceptual understanding of how they work.

Laboratory: Hand-on experience with computer communications, role of communications in modern computing industry, use of variety of tools that require communications e.g. email, internet, WWW and how they work, evolution of mobile computing, demonstration of mobile computing.

(8) **The need to manipulate information (mathematical computations, symbolic computations, intelligence); techniques for simple information manipulation, more complex information manipulation.**

Information manipulation via available packages, word processing, spreadsheets, graphics packages, mathematical computation packages, expert systems. Examples where a package will not do the job.

Laboratory: Use of variety of packages to illustrate how off-the-shelf software can meet a variety of information manipulation needs. Recall word processors for textual and some graphical information, spreadsheets for simple calculations, discussion of conceptual basis of how these tools work, graphics packages for data visualisation, packages for mathematical computations, authoring systems, expert systems, etc.

(9) **The need to specify information manipulation to the machine; introduction to one procedural language; examples of algorithms and their implementations using the language, debugging, testing, and documentation.**

Procedural and non-procedural languages, machine code, assembler, components of a procedural language, introduction to one procedural languages, examples of algorithms, simple programs implementing those algorithms, extension of those algorithms and programs, the importance of debugging, testing, and documenting software.

Laboratory: Discussion of what a language must be able to do to provide instructions to a machine; procedural instructions using machine language, assemblers, and a modern procedural language; example of one non-procedural languages; discussion of algorithms for simple problems, problem solving, study of programs that implement these algorithms, solve a slightly changed problem by modifying the algorithm and the program, discussion of debugging, testing and documentation.

A major disadvantage of this approach clearly is that there is little teaching material available, although the books by Machlup and Mansfield (1983) and Gibbs and

Tsichritzis (1994) provide valuable resource material. The above approach however offers significant advantages that we now discuss. In our view, the approach leads to a much more comprehensive view of computer science with the role of programming more clearly defined. The approach we hope is less time-consuming for the student since it is not driven by programming assignments. Learning of programming takes place at a much more gentle pace allowing the student the time to develop the knowledge, comprehension and application that is required. Also, the approach allows a student who has only limited interest in computer programming to take the core computer science courses perhaps with their other academic interests (e.g. mathematics, visual art). Also, this course could well be more enjoyable for female students given the reduced emphasis on programming. Furthermore, the course allows the students to develop a good conceptual understanding of the basic software tools like word processors, spreadsheets, DBMS, Internet and the Web and some mathematical software and provides them with an understanding of devices that they are likely to have used already (e.g. fax, scanner, CD). A student is also able to build insight by making connections between information processing by computers and information processing in the real world. And finally, as more and more students come to universities after having learned some computing, this course provides them a very different introduction to computing, an introduction that provides a better understanding of the discipline.

4. Courses that Follow

It is not possible in this short paper to present a whole computer science curriculum. We however present a framework that we believe could be followed.

In designing the courses that follow, we use an approach that provides for considerable flexibility. The approach allows for a mathematically inclined student who does not wish to take too much programming to take a number of courses that involve little programming. A student more interested in programming is allowed considerable choice of software engineering projects that should provide him/her an excellent background in software development.

We divide our curriculum in three streams; science of information, design and implementation, and theory and analysis. The science of information courses essentially deal with information representation, storage, retrieval, communication, and presentation but do not include detailed analysis and implementation. Analysis and other theory is covered in the theory and analysis stream of courses while the implementation is covered in design and implementation software engineering group project courses that bring together material from a number of courses and include significant software development.

(1) SI courses - a number of courses in what we call science of information, some of these form the core.

(2) DI courses - a number of design and implementation software engineering group project courses that are optional.

(3) TA courses - a number of theory and analysis courses that are optional and require good mathematical preparation.

The above classification allows a number of different groups of students to follow different curriculum as follows:

(a) *Service teaching*: students who have no deep interest in computer science but wish to learn some computing and some computing tools take the core courses but do not need to take courses in theory or design and implementation.

(b) *Mathematically inclined non-majors interested in computer science*: Such students take the core courses and may take some theory courses. They do not need to take any design and implementation courses.

(c) *Computer Science majors without sufficient mathematics background*: Such students take the core courses and the design and implementation courses but do not take the theory courses.

(d) *Computer Science majors with sufficient mathematics background*: Such students take the core courses, the theory courses and also the design and implementation courses. This is the background that should be required for students who wish to continue to postgraduate studies in computer science.

We now briefly discuss some of the courses.

4.1. SI Courses

A number of SI courses need to be developed and some of them should be part of the core that every student should be required to do. Let the courses be labelled SI1, SI2, SI3, etc.

SI1 has already been presented in the last section. SI2 and SI3 may consist of the following:

Information storage and retrieval - memories, data structures, databases and artificial intelligence.

SI4 and SI5 may consist of the following

Information input/output, human machine interaction, graphics, image processing ??

SI6 may consist of the following:

Computer architecture

SI7 may consist of the following:

Information communication, information security network, parallel processing, compilers

4.2. DI Courses

The design and implementation software engineering project courses essentially deal with implementation of techniques that have been covered in the core SI courses. Each DI course should be provided more staff resources than each of the SI or TA courses. Perhaps each DI course could consist of two lectures per week and a four-hour practical session each week compared to three lectures and a 2-3 hour tutorial each week. Each course will consist of one major group project and may have some much smaller individual assessment items. At least one DI course is proposed for each year in a three-year degree program.

The course DIP1 may consist of the following:

> Procedural programming, design and implementation of a group project.

DIP2 may consist of the following:

> Techniques and Tools - programming languages, software engineering, object-oriented programming etc. Group project involving data structures and algorithms and/or database management system.

DIP3 may consist of the following:

> A group project dealing with one of several aspects of computer science e.g. artificial intelligence, graphics, image processing, etc.

4.3. TA Courses

A number of courses, perhaps two or three, will be needed in this area. The topics to be covered include the following:

> Limits and complexity of information manipulation, analysis of algorithms, computability, finite state automata, formal languages, grammars etc

5. Conclusions

Given the dramatic changes in computing and the computing industry, a question that must be asked is "are the traditional curricula becoming obsolete and have they perhaps outlived their usefulness?". We are inclined to believe that they have and that is why we propose an alternate approach to designing computer science curriculum. We have presented an introductory computer science curriculum that is significantly different than any proposal that we know. The proposed curriculum is, in our view, exciting since it approaches computer science from an information focussed point of view which enables a much broader view of computer science to be presented in the first course.

The new approach has a number of other advantages. The proposal defers and isolates major programming to the design and implementation group project courses where the primary focus is on implementation and where we believe sufficient staff resources ought to be provided to guide the students to good implementations. This approach we believe reduces the frustration and workload in the first course allowing the student to focus on the conceptual basis of the discipline. The introductory course uses computer as a tool which is likely to be more interesting to students rather than a machine which must be tamed, something that perhaps the male students enjoy more. The introductory course also provides conceptual understanding of a number of useful computing tools e.g. word processors, spreadsheets, Internet and the Web, mathematical software and DBMS. Furthermore, the course provides the students an understanding of devices that they are likely to have used already e.g. a CD, video, fax, scanner etc and makes a connection between information processing by computers to that in the real world.

We recognise that our proposal is not without a number of weaknesses. Firstly, it has not been tried anywhere and it is therefore difficult to predict the reaction of students to such a radical shift in curriculum. We suspect at least a number of students who are very keen to get on with the ’real computing’ (i.e. programming) may well be disappointed but we are hopeful that even these students would find the group project environment early in the program more interesting compared to the small programming assignments normally given in the introductory courses.

Acknowledgements

A number of people have helped me in putting these ideas together. It is a pleasure to thank Reinhart Gillner, Shyam Kapur, Bruce Litow, Tony Sloane, Rodney Topor, Chris Wallace, and David Wessels.

6. References

(1) ACM Curriculum Committee on Computer Science, Curriculum 68: Recommendations for the undergraduate program in computer science, Communications of the ACM, Vol 11, No 3, pp 151-197.

(2) ACM Curriculum Committee on Computer Science, Curriculum 78: Recommendations for the undergraduate program in computer science, Communications of the ACM, Vol 22, No 3, pp 147-166.

(3) O. Astrachan and D. Reed (1995), AAA and CS1: The Applied Apprenticeship Approach to CS1, ACM SIGCSE Bulletin, Vol 27, No 1, pp 1-5.

(4) D. Bagert, W. M. Marcy and B. A. Calloni (1995), A Successful Five-Year Experiment with a Breadth-First Introductory Course, SIGCSE Bulletin, Vol 27, No 1, pp 116-120.

(5) D. Baldwin, G. Scragg and H. Kooment (1994), A Three-Fold Introduction to Computer Science, SIGCSE Bulletin, Vol 26, No 1, pp 290-294.

(6) A. W. Biermann (1994), Computer Science for the Many, IEEE Computer, February 1994, pp. 62-67.

(7) P. J. Denning (Ed.) (1989), Teaching Computer Science, Communications of the ACM, Vol 32, No 12, Dec 1989, pp 1397-1414.

(8) P. J. Denning, D. E. Comer, D. Gries, M. C. Moulder, A. B. Tucker, A. J. Turner and P. R. Young (1989), Computing as a Discipline, Communications of the ACM, Vol 32, No 1, pp 9-23.

(9) P. J. Denning (1995), Can There be a Science of Information?, ACM Computing Surveys, Vol 27, No 1, March 1995, pp 23-25.

(10) M. V. Doran and D. D. Langan (1995), A Cognitive-Based Approach to Introductory Computer Science Courses: Lessons Learned, SIGCSE Bulletin, Vol 27, No 1, pp 218-222.

(11) S. J. Gibbs and D. C. Tsichritzis, Multimedia Programming, Addison-Wesley, 1994.

(12) L. Goldschlager and A. Lister (1988), Computer Science: A Modern Introduction, Second Edn, Prentice Hall.

(13) J. Hartmanis and H. Lin (1995), Computing the Future: A Broader Agenda for Computer Science and Engineering, National Academy Press, Washington, DC, 1992.

(14) M. Klawe and N. Leveson (1995), Women in Computing: Where are We Now?, Communications of the ACM, Vol 38, No 1, Jan 1995, pp 29-35.

(15) J. Keaton and S. Hamilton (1996), Employment 2005: Boom or Bust for Computer Professionals, IEEE Computer, May 1996, Vol 29, No. 5, pp. 87-98.

(16) E. P. Koffman, P. L. Miller and C. E. Wardle (1984), Recommended Curriculum for CS1: 1984 a report of the ACM Curriculum task force for CS1, Communications of the ACM, Vol 27, No 10, pp 998-1001.

(17) E. P. Koffman, D. Stemple and C. E. Wardle (1985), Recommended Curriculum for CS2, CACM, Aug 1985, Vol 28, pp 815-818.

(18) F. Machlup and U. Mansfield, Eds. (1983), The Study of Information : Interdisciplinary Messages, Wiley.

(19) D. L. Parnas (1990), Education for Computing Professionals, IEEE Computer, Jan 1990, pp 17-22.

(20) N. Pennington and B. Grabowski (1990), The Tasks of Programming, in *Psychology of Programming*, J. -M, Hoc, T. R. G. Green, R. Samurcay and D. J. Gilmore (Eds.), Academic Press, 1990, pp. 45-62.

(21) R. E. Sabin and E. P. Sabin (1994), Collaborative Learning in an Introductory Computer Science Course, SIGCSE Bulletin, Vol 26, No 1, pp 304-308.

(22) G. Scragg, D. Baldwin and H. Kooment (1994), Computer Science Needs an Insight-Based Curriculum, SIGCSE Bulletin, Vol 26, No 1, pp 150-154.

(23) M. Shaw (1990), Informatics for a New Century: Computing Education for the 1990's and Beyond, TR CMU-CS-90-142, Department of Computer Science, Carnegie Mellon University, Pittsburgh, USA.

(24) A. B. Tucker and P. Wegner (1994), New Directions in the Introductory Computer Science Curriculum, SIGCSE Bulletin, Vol 26, No 1, pp 11-15.

(25) A. B. Tucker, Editor, Computing Curriculum 1991: Report of the ACM/IEEE-CS Joint Curriculum Task Force, Available from ACM Press or IEEE Computer Society Press, 1990.