

A NEW APPROACH TO COMPUTER SCIENCE IN THE LIBERAL ARTS

Brendan Burns
University of Massachusetts
bburns@cs.umass.edu

February 9, 2005

Author's Note: This work describes a curriculum developed while the author was a visiting instructor at Bennington College

Abstract

In the following we make the case for a new course for introducing computer science in the liberal arts classroom. The course is related to breadth-first computer science, but is aimed at developing a broad course suitable for non-majors to gain a comprehensive overview of computer science beyond its algorithmic and mathematic fundamentals. One of the aims of the liberal arts philosophy is to provide students with broad exposure to subject which may not be their primary focus. Neither traditional non-majors courses or introductory programming classes fit this goal. In contrast we propose a course structure which aims to survey computer science in one or two semesters while providing students with significant training and experience in programming.

1 Introduction

Introductory undergraduate courses in computer science have traditionally been separated into two distinct categories: CS 1, an introduction to programming and a wide array of non-major survey courses. Beyond the subject matter, the design of these courses furthers the divide between them. CS 1 emphasizes the learning of basic programming principles, but approaches its subject matter with the assumption that the students in the course will become Computer Science majors and take an additional ten or more subjects covering topics in computer science such as networking, operating systems, artificial intelligence and the like. On the other hand non-major electives directly cover the previously listed subjects and others (“How the network works”, “A survey of computer graphics”, etc)

but in doing so provide the students with very little or no actual programming instruction.

A middle road between these approaches is necessary. One of the primary emphases of the liberal arts approach to education is providing students the opportunity for significant exposure to subjects in which they may not major. Further, many small colleges may not have the resources to support a complete computer science major, examples include Reed College and Bennington College. At Bennington, it is impossible to be a computer science major. With a single computer science faculty member, a sufficient number of courses to produce such a major is impossibility. In either case, there is a need for a course which simultaneously introduces students to programming skills, but also provides them with exposure to other topics in computer science. Neither of the introductory courses previously described satisfy these goals.

In the following, an approach to introducing computer science is proposed which is focused on this goal. The course is designed to cover the canonical subjects of CS 1 such as data types, functions, control, repeated tasks and data representation. In each of the subject areas examples used for classroom pedagogy and homework assignments are drawn from the larger field of computer science. These areas include subjects such as cryptography, web clients and servers, graphical user interface (GUI) design, event driven programming, threads, OpenGL and others. Each broader topic is chosen and matched with a traditional programming subject. At times this requires a certain amount of class time be devoted to general explanations which might not be present in a traditional CS 1 class but the result is that students will gain both programming knowledge and insight into a broad array of subjects in Computer Science.

This approach is related to the notion of a breadth-first introduction to computer science [?], but it is focused on developing a course designed for students who are unlikely to continue as computer science majors. While a breadth-first introduction as it is proposed in [?] proposes to “take the material in the first-year courses – the introductory programming sequence and the discrete mathematics courses – and reassemble them” into a more holistic course. The structure proposed herein proposes using real world applications of computer science such as cryptography, 3D graphics, and networking to serve as programming examples and labs.

Several further benefits result from this use of practical examples in an introductory programming course’s pedagogy. By gaining hands-on experience developing solutions to “real world” problems students gain experience in areas in which they may actually work in the future. An art major who learns OpenGL programming has expanded their understanding and ability to create digital art. Additionally it has been my observation that the development of solutions to real world problems has led to greater confidence in their programming skills than developing the solution to toy problems. Student’s perception of the challenge (and therefore the value of their success) is enhanced by creating programs (like a web client or server) which they actually use.

2 Related Work

There are several books which attempt to address similar issues to the ones I have mentioned previously. In particular “How to Design Programs” [2] and “An invitation to Computer Science” [4], are addressed toward a similar audience, though provide significantly different approaches.

“How to Design Programs” provides an elegant introduction to programming and problem solving in the Scheme programming language, it is approachable and suitable for non computer science majors. However, it does not generally approach subjects other than programming in its examples and problems.

At the other end, “An invitation to Computer Science” provides a high-level overview of the computer science discipline, and is specifically targeted at non computer science majors. However, although it offers an exposure to the subject of programming, it is not its central focus. A course built around the book would not leave students with the ability to independently develop programs.

Vandenberg and Wolloski [?] present a single semester breadth-first computer science course, but the domain from which they draw examples for programming projects is largely limited to programming languages, computer hardware and theoretical computer science.

The 2001 report on Computing Curriculum [?] provides a definition for breadth-first computing which incorporates programming and discrete mathematics into an integrated set of three courses. This approach is designed with the understanding that students are computer science majors who will continue in the field. As such it gives little exposure to topics beyond the mathematics of computer science.

3 The Course

Although computer science is often taught in liberal arts colleges, its method of instruction is rarely geared toward the liberal arts experience. The liberal arts experience is characterized by exposing students to a wide range of elective options for their education. In choosing a liberal arts college a student gains the opportunity for exposure to a variety of subjects.

Introductory computer science courses can be (somewhat coarsely) grouped into two categories: Introduction to programming or The network/The computer/ Artificial Intelligence/... for the non-major. Neither of these two types of courses provide a complete experience in computer science for the liberal arts student. In the liberal arts course, a middle road is needed, a course which can teach introductory programming skills while offering an exposure to a variety of topics in computer science. It is such a course that is proposed in the following.

3.1 Goals

The design a course which can give students a comprehensive exposure to computer science begins by outlining the goals of the course. In a sentence, the goal of this course is to introduce students to computer science while simultaneously teaching them how to program. To accomplish this, real-world topics of computer science are integrated into the instruction of programming. Every programming course requires example problems to illustrate subjects such as control, recursion, data structure design and more. In this course, rather than designing toy problems to illustrate these subjects, real aspects of computer science are used.

The use of real computer science examples to teach programming serves a number important tasks in the liberal arts classroom. First, as in any programming course, the examples and problems teach students the fundamentals of programming. Secondly, real-world examples help illustrate and explain the subject area from which they are drawn. Additionally, providing non-computer science majors with practical experience in real world subjects give students programming skills which may be relevant to their chosen major. For example an art major may now be able to develop dynamic web-based pieces, or use computer graphics in an installation. Finally, choosing real-world examples from subjects which relate to student's day to day lives (the web, computer games, encryption, etc.) enhances student interest in the current subject of the course.

3.2 Subjects

The following outlines a course, suitable for a single semester introduction to programming which uses a variety of traditional computer science topics for its examples. The various subjects of a traditional introduction to programming course and the real-world examples which pertain to them are described in detail.

Data Types

Programming begins with the basic data types for a programming language. Data types offer the opportunity to explore the manner in which the computer represents data internally. The exploration of data types begins with the boolean type, a direct analogy for a bit in the machine. With the introduction to booleans, students are also exposed to the logical operations (and, or, exclusive-or and not) for constructing boolean expressions. Following booleans, students are introduced to integers and the binary representation of numbers. Subsequently the character data type offers students the awareness that every type is simply bits, and it is the interpretation, in this case via ASCII or Unicode which provides meaning. Following on this theme, although more complicated mathematically, floating point representation and real valued types are introduced.

Functions

Once data types have been introduced the next step in programming is the

definition of functions. The subject of cryptography provides a compelling source of examples for defining and composing functions. As a simple example, consider implementing a simplified version of RSA encryption. The encrypting/decrypting function is simply:

$$\mathit{crypt}(\mathit{plain}, k, n) = \mathit{plain}^k \bmod n$$

This function is straight-forward to express in any programming language. Verification that the encryption/decryption works is provided by compositing two crypt functions together using the public/private key pair (k, k') :

$$\mathit{plain} = \mathit{crypt}(\mathit{crypt}(\mathit{plain}, k, n), k', n)$$

In addition to teaching students how to implement and composite functions, the subject matter offers the opportunity to explore the number theory behind cryptography, and the practical implementation of public key systems. These subjects are important to computer science and, in my experience, of great interest to students.

Abstraction/Parameterization of functions

Once students have a handle on defining functions, the next step in instructing them as programmers is the design of abstract, general purpose functions. For a domain for this subject we turn to OpenGL and 3D rendering. It might seem that this subject is far too advanced for an introductory course, however, with helper code to handle the details, programming graphics is as simple as drawing lines between coordinates in a three-dimensional environment. As an example, the commands to draw a pyramid in OpenGL are shown in Figure 1.

How can graphics illustrate abstraction? Students begin by defining functions such as the one to draw a pyramid. The function initially draws a pyramid of a fixed size centered around the origin. This function takes no parameters. Next students abstract the function so that it can draw a pyramid at an arbitrary (x, y, z) coordinate. Finally, students abstract the function further so that the scale of the pyramid in each of the dimensions can be controlled as well. Once written in this way, the function can be used to draw a myriad of pyramids. Through this process, students learn the value of abstract functions and a method for developing them.

Along with encryption, computer graphics are of great interest to a student for obvious reasons. With the proper helper functions to create windows and the context for 3D rendering, constructing simple shapes is well within the abilities of students in an introductory class and creating these 3D worlds develops confidence and insures student attention to the coursework.

Control/Recursion

Having mastered functions, the next subject of an introduction to programming is to control, loops and recursive functions. For an example, students

```

# front
glVertex3f 0 1 0
glVertex3f 1 0 1
glVertex3f -1 0 1
# right
glVertex3f 0 1 0
glVertex3f 1 0 1
glVertex3f 1 0 -1
# back
glVertex3f 0 1 0
glVertex3f 1 0 -1
glVertex3f -1 0 -1
# left
glVertex3f 0 1 0
glVertex3f -1 0 1
glVertex3f -1 0 -1

```

Figure 1: Drawing a Pyramid in OpenGL

```

GuessingGame(min, max) : number
  guess := (min+max)/2
  if(correctAnswer(guess))
    return guess
  else if (lessThanAnswer(guess))
    return GuessingGame(guess, max)
  else
    return GuessingGame(min, guess)

```

Figure 2: The algorithm for the recursive divide and conquer guessing game

develop a program which plays a numbers guessing game. In addition to illustrating a simple artificial intelligence, the solution to the number guessing is a great example of a divide and conquer algorithm, the recursive solution to which is quite simple (Figure 2). The problem can also be solved using a loop (Figure 3. In addition to introducing the students to a simple example of artificial intelligence, loops and recursion, this example serves to illustrate the central fact of computer science: “There’s more than one way to do it.” It provides an opportunity to discuss both programming style and the role that programing choices make on the performance of an algorithm.

Functions as first class objects

The notion of a function as a first class type and passing a function as an argument is present in nearly every programming language, in varying complexity. Since this course was first developed with the Scheme programing

```

LoopGuessingGame(min, max) : number
  guess := (min+max)/2
  while(!correctAnswer(guess))
    if(lessThanAnswer(guess))
      min=guess
    else
      max=guess
    guess := (min+max)/2
  return guess

```

Figure 3: The algorithm for the looping divide and conquer guessing game

Node	Arc
arcs : list	input : char
terminal : boolean	next : Node

Figure 4: The data structures for defining deterministic finite automata

language where functions as arguments are straight-forward, an example of this programming topic is apropos. Event driven graphical user interfaces (GUIs) provide an excellent example of this topic. Functions are passed as handlers for mouse clicks, button presses, menu selection, etc. This provides a great example of passing functions as arguments and an introduction to the programming of GUIs. The applicability of this subject to introductory programming has already been proposed [1].

Data Representation

Once students can define sophisticated functions, the next step in introducing them to programming is the representation of data within programs. While data structures is (and should be) a course unto itself, the introduction of data design to students is important enough to be mentioned in an introductory course as well. To illustrate the design of a data representation, deterministic finite automata are used. A representation of a DFA can be developed using two data structures a node and an arc. The node simple contains a list of arcs and whether or not it is an accepting state. An arc contains the input that it consumes and the state that it leads to. These data structures are illustrated in Figure 4. Given these structures, it is simple to write an algorithm for operating the DFA on some input string. The use of DFA introduce students to a fundamental topic in theory of computation, as well as introducing them to the notion of regular expressions which are widely used for parsing textual input. The discussion of parsing can serve to illuminate a portion of how a high-level programming language (or even free text) is understood by a computer.

```

WebServer()
  do forever
    request := waitForRequest()
    thread := new Thread(HandleRequest, request)
    start(thread)

```

```

HandleRequest(request)
  file := parseRequest(request)
  responseHeaders(file, request)
  byte := read(file)
  while (byte is not EOF)
    write(byte, request)
    byte := read(file)

```

Figure 5: The algorithm for a simple file-based web server

Input/Output

Input/output is a fundamental topic of both computer science and introductory programming. A first example of this topic is file encryption. The encryption function developed in the first example is combined with simple byte reading and writing to implement file encryption. A second example, providing an introduction to network programming and protocols is an implementation of a web client. The HTTP protocol and stream abstractions present in most programming languages make this an appropriate problem for an introduction to programming course.

Threads

While the subject of concurrent programming is often left to more advanced classes. The abstraction of threads in languages like Java and Scheme mean that the subject is approachable and appropriate for an introductory student. A great example of threaded programming and a good follow-on for writing the web client, is the implementation of a simple web server. This might seem like a daunting task for an introductory class, but in truth the code (with the assistance of some helper functions) is quite simple. The pseudo-code for the server is given in Figure 5. The functions `waitForRequest`, `parseRequest` and `responseHeaders` are helpers functions provided to the student. This project also provides a nice conclusion to the programming course, allowing students to understand via implementation a type of program they use on a daily basis.

4 Experience and Conclusions

In the fall of 2003 and the spring of 2004, I taught two introductory courses based on this course at Bennington College in Bennington, Vermont. The courses were taught in the Scheme language using the DrScheme [3] environment. Scheme and DrScheme in particular serve as an ideal environment for teaching this course. Scheme provides a number of high-level abstractions, for example in file and network I/O, which make programming easier. The DrScheme programming environment provides high-level access to GUI widgets and OpenGL graphics. Additionally, the TeachPack infrastructure enables students to easily gain access to helper functions which wrap complicated details of things like the HTTP protocol.

The students who took my course had varying degrees of experience with computers but none had any prior programming experience. I found that the course was adequate to both tasks for which it was designed, namely the development of programming skills and a survey of a variety of topics in computer science.

Subjects like encryption, 3D graphics and the web client/server allowed students to explore and begin to understand technologies they use on a daily basis. The classes on these subjects also offered the opportunity for students to ask questions about the subjects and gain additional insight into the topic as an area of computer science. Additionally, an unexpected benefit of the course was greater gains in student confidence about their skills than I expected. I believe that because the students were building programs they perceived as “real” they developed greater confidence in their abilities. The development of this confidence on the part of the student in their ability to attack and solve problems is one of the most important skills to develop. Instilled with confidence, students are willing to explore, believing that they will eventually find a solution. Too often I have seen students in introductory programming courses lack the confidence to even begin to attack a problem.

Introductory computer science courses are currently divided into introductory programming classes and courses for non-majors. This artificial division is particularly inappropriate in the liberal arts course, especially at small schools like Bennington where a computer major is not an option. In the proceeding I have proposed a middle road, an introductory programming course which simultaneously surveys a variety of topics in computer science. While I have proposed this class in the context of the liberal arts course, I believe that it is appropriate to any situation where students may be seeking an in-depth survey of computer science without the requirement that they become majors.

References

- [1] ACM. Computing curricula 2001. <http://www.computer.org/education/cc2001/final/>.

- [2] K. B. Bruce, A. Danyluk, and T. Murtagh. Event-driven programming can be simple enough for cs 1. In *Proceedings of ITiCSE*, 2001.
- [3] M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi. *How to Design Programs*. MIT Press, 2003.
- [4] PLT. The drscheme programming environment. <http://www.drscheme.org>.
- [5] G. M. Schneider and J. Gersting. *An invitation to Computer Science*. Course Technology, 2000.
- [6] S. Vandenberg and M. Wollowski. Introducing computer science using a breadth-first approach and functional programming. In *Proceedings of ACM SIGCSE*, 2001.