A More Formal Approach to "Computer Science: Principles"

Rex Page
University of Oklahoma
Norman, OK, USA
page@ou.edu

Ruben Gamboa
University of Wyoming
Laramie, WY, USA
ruben@uwyo.edu

ABSTRACT

We report on a course, entitled "How Computers Work: Logic in Action", which we have offered the past few years at the University of Oklahoma, and which will be offered soon at the University of Wyoming. Intended for non-CS majors, this course is our answer to the question, *What would you teach if you had only one course to help students grasp the essence of computation and perhaps inspire a few of them to make computing a subject of further study?* Assuming no prior knowledge of computers or mathematics beyond high school algebra, the course is compatible with the *Computer Science: Principles* approach proposed by the College Board, although it is a significant departure from the pilot courses that are currently following this approach.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer and Information Science Education; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*Mechanical theorem proving*; D.1.1 [**Programming Techniques**]: Applicative (Functional) Programming

Keywords

First year, service course, formal computation.

1. BACKGROUND

For the past few years, we have had the opportunity to teach a broad computer science course to honors students at the University of Oklahoma. The corps of honors students is a self-selected group, with above average abilities, motivation, or both. To qualify for the honors program, students must hold a grade point average (GPA) of at least 3.4 out of 4.0 when they join the honors program, and they must maintain this GPA at the time of graduation to receive an honors designation in their diploma. Because of their high academic engagement, these students can succeed in almost any course, so they make our job as professors easy. However, their diverse backgrounds also make them particularly challenging: 44% of the students who have taken our course are science majors, ranging from physics and chemistry to meteorology; 31% are engineering majors, including computer science and computer engineering, as well as many traditional engineering disciplines; and 22% are humanities, social sciences, or business majors. Another measure of the students' disparate backgrounds is provided by their computing experience: 60% had some prior exposure to programming, primarily in high school, but the others knew very little of computing. One place where the students were similar to each other was in academic classification: nearly all of them were in their first two years of college.

So our challenge was clear: How to design a course that presents an overview of the essential principles of computation to a group of talented students with varying degrees of computing experience and mathematical preparation. This immediately presented us with two seemingly conflicting challenges. Since the students were majoring in many different disciplines, the only guarantee we had with regards to mathematical background was the minimum required by the university, namely high school algebra. So it was imperative that the course be accessible to a student with only this minimum background. Of course, this also means that the course is accessible to many, if not most, high school students. And this leads into the second challenge. While the course must remain accessible to students with little mathematical preparation and no prior exposure to computer science, it must also be engaging and challenging to those students who have had some programming experience.

But this only masks the deeper issue. What exactly are these "essential principles of computation" that we wish students to learn? Our answer to this question is influenced by the First Year curriculum proposed by Felleisen and the PLT group[7, 8], but adapted and pared down so that it can be fruitfully presented in a single semester to students who may not take another computer science course. In particular, the First Year blends program design, discrete math, and the beginnings of software verification using the automated theorem prover ACL2, and all these are also features of our course [12, 6].

Here are the principles we wanted to impart to students. To start with, students should build a mental model of computation, so they can see how it's possible for mechanical devices to perform computations. So one essential principle is as follows:

1. Algebraic formulas can specify computations.

Once students understand that computations can be specified using ordinary algebra, it is possible to discuss how software systems can be constructed. This leads to the next essential principles:

2. Abstractions allow solutions to small problems to be arranged into the solution to big ones.
3. Important, complex algorithms derive from simple, definitional properties.

The relevance of these ideas is made clear by considering some applications of software. Our goal is to make sure that students see the connection between software systems—specified as sets of algebraic equations—and the real world. We also stress the importance of getting the software right, precisely because the software is solving real-world problems.

4. There is a strong correspondence between logic formulas and digital circuits.
5. Models expressed in software capture the behavior of processes and devices.
6. Testing and reasoning critically about software are important tools in ensuring that the software works as intended.

The relevance of the material is supported even more by considering applications familiar to students, e.g,. the internet. What we want the students to understand is that computers work on the concept of *scale*. The big applications they are familiar with operate on the exact same concepts they learn in class. The only difference is that the problems are bigger, so abstraction plays a major role, and the final solution is composed of many, many steps. In other words, the main difference between computing as they see it in class and computing as practiced in, say, Google, is one of scale. Solving the scale problem is an engineering problem, the main problem studied in a computer science degree.

7. Different definitional properties can specify the same functions, but at vastly different computational expense.
8. Computational expense makes some useful devices feasible and others infeasible.
9. All of the ideas in the course bear on the ability of computers to deal with information on the massive scale needed to provide services like search engines, internet storefronts, and social networks.

This is not the typical introductory computer science course! The material does not overlap directly with the mainstream material from a computer science or computer engineering program, so those students with prior exposure to computers find it new and challenging.

A feature of this course is that it includes topics from logic and software verification, which are usually not included in the computer science curriculum. We believe that the course has been successful because of the algebraic description of computation that we use. In this view, programs are really ordinary equations, very similar to the equations students have seen in high school algebra. So the students learn about computation in a comfortable framework. Moreover, this framework lends itself to automatic verification, and students learn how to use the theorem prover ACL2 to automate many of the reasoning tasks. ACL2 is not completely automatic, usually requiring much assistance from its users. However, the necessary assistance can be minimized by carefully choosing the programs to study and the properties under consideration, so that the proofs of these properties can be readily found using ACL2's heuristics. Others who have also incorporated ACL2 into lower division computer science courses have also found that students can benefit from using ACL2 for the purposes of the course, without having to become ACL2 experts [6, 2]. The end result is that students learn how to think formally about computation, without being overwhelmed by the logical machinery that would normally be required.

In its approach and objectives, this course has much in common with the *Computer Science: Principles* course proposed by the college board [15]. For instance, our course is not designed to teach students how to program. Students do learn some programming, but not enough to take on any serious development project. Instead, the course tries to provide insight into the nature and relevance of computation.

Our course differs significantly from other courses in the *Computer Science: Principles* vein. Currently, a number of pilot courses are running nationwide [16]. The majority of these use scripting languages to facilitate programming. Python is popular for this purpose, as are visual scripting languages like BYOB (aka SNAP!) and App Inventor [10, 9]. Doing so has a major advantage, in that it allows to create software that is very similar to what they use daily, e.g., App Inventor allows them to build mobile applications with only minimum effort. It also allows more time to be spent on the social aspects of computing, so many of these courses feature topics from privacy and security, as covered in the textbook from Abelson et al [1]. There are many things we like about this approach, but we chose to go in a different direction because we wanted students to really understand the layers of abstraction that allow simple ideas from logic, which are readily implemented as circuits, to implement complex applications, e.g., mobile apps or popular websites. And we also wanted to emphasize how mistakes in software can be avoided and the importance of doing so, since software is increasingly relevant in our lives in many different contexts.

We will address the connections between our course and the principles approach in Sect. 3. But first, we will describe the course in more detail in Sect. 2, and we share the results from the previous offerings of this course in Sect. 4.

2. COURSE CONTENTS

We use equations extensively, both to describe functions and to present properties that these functions should have. Equations are a good fit for a computational approach based on algebra, but they are not the usual way in which software is described. For example, the imperative model of programming introduces variables to hold state, assignment statements to mutate state, and other programming constructs to control the execution of the program. Consequently, we chose to use a purely functional approach to programming, and in particular we use the language ACL2, which is a variant of Common Lisp. To our knowledge, undergraduate students are only exposed to ACL2 in a few courses nationwide. But what makes it ideal for our purpose is that it is not just a programming language. It is also a logic of computation and an automated theorem prover for this logic. So students learn the basics of programming *and* of reasoning about such programs.

We introduce the equational style of programming with a handful of examples. For example, the classic Lisp list operations can be introduced with the following equations:

$$\begin{array}{lcll}
(\texttt{cons}\ x\ [x_1\ x_2\ ...\ x_n]) & = & [x\ x_1\ x_2\ ...\ x_n] & \{cons\} \\
(\texttt{first}\ [x_1\ x_2\ ...\ x_{n+1}]) & = & x_1 & \{fst\} \\
(\texttt{rest}\ [x_1\ x_2\ ...\ x_{n+1}]) & = & [x_2\ ...\ x_{n+1}] & \{rst\}
\end{array}$$

Note that the equations are labeled, so that we can refer to them later.

Students have no problems understanding these equations, and they quickly gain an understanding of what these functions are. In fact, they readily accept (without any sense of proof) that the following two equations must also hold:

$$\begin{array}{lcll}
(\texttt{first}\ (\texttt{cons}\ x\ [x_1\ x_2\ ...\ x_n])\ ) & = & x & \{fst\text{-}id\} \\
(\texttt{rest}\ (\texttt{cons}\ x\ [x_1\ x_2\ ...\ x_n])\ ) & = & [x_1\ x_2\ ...\ x_n] & \\
 & & & \{rst\text{-}id\}
\end{array}$$

Even at this early stage, we can introduce one of the main ideas of the course, namely that testing and critical reasoning are necessary to ensure that the functions work as expected. Students certainly believe the last two equations—but are they actually true? To verify this, we make use of the `DoubleCheck` testing framework of Dracula [6, 5]. In the style of Racket (formerly known as DrScheme), Dracula is a front-end that makes it easier for many students to interact with ACL2. One of its most useful features is `DoubleCheck`, which was inspired by the `QuickCheck` testing framework for Haskell. It allows students to specify properties that should be true of either built-in functions or new functions they write. For example, a student could submit the property *fst-id* to `DoubleCheck`, which would then generate random tests to see if the property is likely. If the tests pass—and in this case, they surely will—the library moves on to the second ("double") check. I.e., it submits the property to the ACL2 theorem prover, which tries to prove it automatically. In the case of *fst-id*, the theorem prover is able to verify the property automatically, so the student can be assured that the property in fact holds for all inputs.

The equations *fst* and *rst* capture the essence of `first` and `rest`, but neither equation fully captures the corresponding function. For example, what is `first` of the empty list? The actual answer to this question is not important, and different programmers (and even programming languages) will provide different answers. But what *is* important is that there be some answer. E.g., we may choose to use the following equation to resolve the matter:

$$(\texttt{first}\ [])\quad =\quad []\quad \{fst\text{-}empty\}$$

Thus students are introduced to the concept of function definitions. In this model, functions are defined by a set of equations. We stress that the equations must satisfy two key properties. First, they must be *consistent*. That is, no two equations can provide different results for the same input. Second, they must be *comprehensive*. All forms of input must match the left-hand side of at least one equation.

Things become more complicated when inductive (also called recursive) definitions are necessary. Opinions certainly vary as to whether recursion is a topic suitable for beginning programmers [13, 17, 3, 11]. However, we find that inductive definitions arise naturally in the context of properties. To see this point, consider the square root function, which is very familiar to all students. They would certainly believe that $\sqrt{x \cdot y} = \sqrt{x}\sqrt{y}$ is true, and nobody would remark that this is an inductive property. Of course, this one equation is not a definition of square root, but that is precisely the point. Properties that use the same function symbol in the left and right-hand sides pose no immediate problem, and *some* inductive properties can be used to define functions.

In fact, students readily accept the following properties for the `append` function:

$$\begin{array}{lcll}
(\texttt{append}\ (\texttt{cons}\ x\ xs)\ ys) & = & (\texttt{cons}\ x\ (\texttt{append}\ xs\ ys)) & \\
 & & & \{app1\} \\
(\texttt{append}\ [\ ]\ ys) & = & ys & \{app0\}
\end{array}$$

In the beginning, we encourage students to think of these equations simply as properties that any reasonable definition of `append` must satisfy. Once the students are comfortable with these equations, we extend the notion of function definition—that is, definitions by way of equations—to include inductive definitions. As before, such equations should be *consistent* and *comprehensive*, but now we add a third criteria for a valid definition. The equations must be *constructive*, i.e., any inductive reference to an operator must be on a reduced computation. In the case above, the inductive reference to `append` is on the value of xs which is smaller than `(cons` x xs`)` in the left-hand side.

What students learn is that all properties of any function derive from a set of equations, as long as these are *consistent*, *comprehensive*, and *constructive*. These three characteristics of definitional equations, which we refer to as "The Three Cs," form a recurring theme throughout the course.

Students analyze functions defined inductively in the same way as they do other functions. For instance, students (or the instructors) can state properties that they believe the functions should hold, such as the associativity of `append`. These properties can be expressed using the `DoubleCheck` testing framework, and Dracula will test the properties on random values. If the tests pass, the property is then submitted to the ACL2 theorem prover for formal verification. The difference is that in this case the proof requires induction.

At this point, students mostly believe that all properties of the function `append` must follow algebraically from its defining equations. We reinforce this concept now by demonstrating how the associativity of `append` can be proved formally. A pencil-and-paper proof of this property is based on induction on the length of the list xs. The base case uses the defining equation $\{app0\}$, while the inductive case relies on $\{app1\}$ and, of course, the inductive hypothesis. While students do not necessarily learn how to carry out these proofs by themselves, they come to appreciate that the same algebra they learned in high school, albeit with less familiar functions, such as `cons`, can be used effectively to think about software and to guarantee that a particular function works as expected.

It should be noted that proofs of basic facts, such as the associativity of `append`, can be quite challenging. Students do learn how to write down some of these proofs, and they certainly learn how to read them. Mostly, however, they use ACL2 to find proofs of the more substantial theorems. Here, too, the students can run into trouble, because even though ACL2 is fully automated, it does not always find proofs of even basic theorems. This means that students must also discover (with some help) key lemmas that can be used to guide ACL2 to the desired results. To minimize the students' frustration, we have designed specific problems that work exceptionally well with ACL2's heuristics, so that many proofs are found fully automatically.

In the remainder of the course, students gain more experience with programming using equations to define functions. Our approach is to introduce programs in a variety of small application domains, and to ask students to write (or complete) some of the functions that are useful in that domain.

Of course, discovering equations satisfying "The Three Cs" that can define a particular function is no easy task! This requires more than an understanding of algebraic rules. We tell students, and they soon agree, that it requires insight and creativity, even though defining functions entails "nothing more than" working with ordinary, algebraic equations and classical logic. In fact, this is another recurring theme in the class. Simple mechanisms can combine to produce extremely complex results, sometimes by accident, and sometimes not—and doing it intentionally is profoundly creative.

The first application domain that we use as a programming playground is the world of propositional logic and combinational digital circuits. This domain is advantageous, because the equations of Boolean algebra are very similar to the familiar equations from high school algebra. We show students how all properties of Boolean algebra, including the traditional truth tables, can be derived from a handful of equations that suffice to capture the meaning of the propositional connectives. This gives students practice in the type of algebraic reasoning that they will later employ when reasoning about more complicated functions.

After introducing propositional logic, we move on to digital circuits. This serves two distinct purposes. First, it allows us to directly connect what the students have been doing in the course so far to working, physical computing devices. Second, and more important, it begins an explicit discussion on the nature of abstraction: propositional formulas and combinational circuits are quite obviously related, and students can analyze one of them by studying the other.

Next, we provide a brief introduction to computer arithmetic, culminating in a ripple-carry adder circuit. Again, this provides us with an excellent opportunity to discuss abstraction. At the top level is ordinary arithmetic, which the students take for granted. Below that are the rules of numerals, i.e., lists of digits and arithmetic operators on those lists. Numerals are a representation for numbers, and different strings of numerals can represent the same number. This point is driven home by considering the binary representation of numbers. The process of abstraction continues, because binary digits themselves represent different voltages in wires of digital circuits. The result of all this is that students see how it is that physical devices can perform meaningful computations. I.e., they can see how a particular circuit consisting of logical gates can seem to compute the sum of two numbers. More than that, the different layers of abstraction are precisely defined, and the transitions between these layers can be formalized by a set of equations which be checked with `DoubleCheck` and verified informally with hand proofs and formally with ACL2. This culminates in the following ACL2 theorem, which is presented in class:

```
(implies (and (bitp c0) (bit-listp x) (bit-listp y)
              (= (len x) (len y)))
         (let* ((a (adder c0 x y))
                (s (first a))
                (c (second a)))
               (= (num (append s (list c)))
                  (+ c0 (num x) (num y)))))
```

This theorem uses the "type-checking" functions `bitp` and `bit-listp` which recognize bits and lists of bits, respectively. It also uses the function `num` which converts a list of bits into a number, and `adder` which adds two lists of bits encoding numbers in two's complement notation. All of these functions were previously defined in our equational style.

This process continues with the introduction of some typical data structures, such as trees and hashes, and different combinational circuits, such as multiplexers. As the problems get bigger, we provide the students with more and more guidance and supporting artifacts, so that they can continue learning without having to write (and reason about) an inordinate amount of code. Eventually, we reach the top of the abstraction pyramid in a very high-level discussion of computing systems that students use in their daily lives. Here the discussion is at the conceptual level, and not at all at the nuts-and-bolts level of a ripple-carry adder.

The first major application comes from Facebook. Throughout the course, we have been convincing students that simple mechanisms can combine to produce extremely complex results, and that the same basic mechanism they learn about in class have been put together in creative ways to produce all the computing artifacts they have already seen. The only mystery is how to deal with scale, both the scale of the program itself and the scale of the data it works on—and Facebook is a perfect case study on scale.

We tell the students that Facebook must combine data from two large datasets in order to produce a single page. It must consult its dataset of "friends" so that it knows which status updates to post, and it must also ccons-first-rest-defonsult its dataset of "statuses" so that it knows what your friends have been saying. Without providing any evidence, we tell them that combining these two datasets is conceptually simple but computationally infeasible with traditional technology because of the sheer number of facts. As a consequence, Facebook developed its own database solution, the NoSQL database Cassandra [14]. We do not expect students to learn much about databases, whether relational or NoSQL in this course. However, students can grasp the essence of Cassandra replication, and they can readily see how this affords the massive scalability required by Facebook. Students also see how Cassandra uses many of the algorithms and data structures they have already seen, such as hash functions and lists.

Next, students learn a little about scaling at Google. Again, the need for scale is easily motivated because of the sheer size of the web. We ask students to consider the difficulty of doing something as simple as finding how many other web pages link to each page in the web, a key metric in the PageRank algorithm that Google uses for ranking search results. The answer, we show them, is Google's distributed platform MapReduce [4]. With its roots in functional programming, MapReduce is actually easy to introduce to students who are already familiar with Lisp in the form of ACL2, so we can present relevant examples with actual code.

For example, WordCount is one of Google's simple MapReduce examples. In this case, the input to the map function is a word appearing in some document and a partial count, which can simply be 1. Our initial sample consists of each word from the Gettysburg Address and an initial count of 1. In general, a main program would process a document and produce such calls to map. The output of map is a list of partial word counts. Only one word (the input word) appears in this list, but the MapReduce framework requires that the function return a more general list of values. The inputs to the reduce function are a word and a list of all the counts that were seen for this word. The output is a list of words and their combined sums, computed using the auxiliary function `sumlist`—again, there is only one word in the list, although the extra container is required by the MapReduce framework.

$$\begin{aligned}
(\texttt{wc-map}\ w\ c) &= [\ [w\ c]\ ] && \{\textit{wc-map}\}\\
(\texttt{wc-reduce}\ w\ cs) &= [\ [w\ (\texttt{sumlist}\ cs)]\ ] && \{\textit{wc-reduce}\}\\
(\texttt{sumlist}\ []) &= 0 && \{\textit{sumlist-0}\}\\
(\texttt{sumlist}\ [x_1\ x_2\ ...\ x_n]) &= x_1 + (\texttt{sumlist}\ [x_2\ ...\ x_n]) && \{\textit{sumlist-0}\}
\end{aligned}$$

3. CONCORDANCE

Although our course is very different than the official pilot courses for the *Computer Science: Principles* initiative, we believe that it can be used in that setting [16]. In this section, we explore how our course satisfies the theme and topics covered in *Computer Science: Principles* (henceforth, CSP).

For instance, the CSP contains seven "big ideas," further divided into specific learning objectives and evidence that supporting them. The first big idea is that computing is a creative activity, and the associated learning objectives include understanding that tools and techniques from computer science are used to *create* and analyze computing artifacts, but it goes beyond that. It also involves using computing as a means for creative expression and as a tool to build artifacts that have some practical applications (even if only for personal reasons.) This idea is central to our course (as to many other introductory and advanced courses in computer science.) Students have the opportunity to create several programs *and* to analyze many of their properties, mostly of the correctness flavor, e.g., "the circuit for a ripple-carry adder correctly performs addition of n-bit binary integers." Students readily agree that writing programs that can perform specific tasks is an intensely creative process. Moreover, students are exposed to a number of different application domains and they see how very practical problems, such as solving arithmetic problems or indexing the world-wide web, can be solved using computation.

The second big idea is abstraction, which comes in a variety of flavors. In our course, abstraction is one of the main ideas. This is the reason why we present boolean expressions and combinational digital circuits, and why we go on to build machine representations of arithmetic. The students quickly appreciate the benefit of abstraction in information hiding. In addition, students explore properties of the various abstractions and mappings between them, and they learn that some properties of a complex model can be more easily explored by analyzing a simpler one.

The third big idea involves data. Students are expected to use computers to explore data sets and understand how the different ways in which the data can be represented affect the exploration. Our course does expose students to this idea, although we have not had the time to explore it as deeply as other CSP courses. We do stress that the big difference between toy applications and real-world applications is a matter of scaling, both in data and program size. And we show students examples of how large data sets are used and processed, e.g., by Google and Facebook.

The next two big ideas encompass algorithms and programming, and both of these are very much at the center of our course. We describe algorithms using algebraic equalities, and this is something that students start doing from the first few classes. Students learn to develop their own algorithms in different applications, and they learn to express these algorithms in ACL2, which as a programming language is almost identical to the pure functional subset of Common Lisp. Students also evaluate these algorithms, mostly in terms of correctness, but also in terms of performance, and they learn that some algorithms yield efficient computations while other algorithms to the same problem result in infeasible computations. This last point is especially central to the discussion of data structures, e.g, when describing binary trees, balanced trees, and hashtables.

The sixth idea is concerned with the internet, and it includes knowing basic networking concepts, characteristics of the internet, and basic concepts from cybersecurity. Our course does address the internet, but not as much as other CSP courses. In particular, we discuss the structure of the web and the notion of indexing and ranking as it relates to search engines. We also discuss different types of web applications, including traditional Web 1.0 storefronts and the more social Web 2.0 applications. In addition, we describe how web applications are implemented and discuss specific components of the solution, such as web servers and databases, but without going into any depth.

The seventh and final big idea concerns the global impact of computers. This actually fits in very well with our approach, because the course is currently directed at honors students, who bring a diverse background to the course, including prior exposure to courses in the humanities. The impact of computers, both good and bad, is a topic for discussion in class. More important, it is a topic that students explore more fully in a research paper and class presentation.

4. OUTCOMES AND CONCLUSION

We have offered this course twice at the University of Oklahoma, and it is scheduled for the current academic year. As of this writing, we are also in the process of migrating the course to the honors program at the University of Wyoming, where it will be offered for the first time this year.

This is not an easy course. Although the material is couched in the familiar framework of algebraic equations, it includes topics in logic, digital circuits, programming, testing, verification, and selected topics from data structures, algorithms, and other areas of computer science. Nevertheless, the course is accessible to students who have had only the standard college preparatory work, in particular high school algebra.

What we found is that all students do well in the course, despite their different backgrounds. The engineering students outperformed the students in the arts and sciences, but not by much. And the difference between students in the arts and those in the sciences was negligible.

Students have been satisfied with the course. They found the course challenging but rewarding. Their anonymous comments suggest that they gained a fundamental understanding of what computers can actually do and how they can do such things. This is, in our opinion, the basis for computational thinking.

Because the course has been offered under the framework of the honors program, we have another way to judge student satisfaction. Every year, different honors courses are proposed, and the students have an important voice in determining which of these courses are actually offered. Our course has been selected three times now, and placed prominently as one of a handful of options that can satisfy one of the major requirements for the honors program.

When we started this effort, we believed that it was possible to offer a course that could introduce students from many different backgrounds, not just computer science majors, to formal ways of thinking about programs. We also believed that such a course could demystify computers and place them in a larger historical context. However, we were concerned that students would find the mathematical rigor to be too challenging. We can now report that the more optimistic viewpoint was the correct one, and we are convinced this is due in large part to the presentation of computation as a form of equational reasoning and the use of tools to aid in proof discovery. Specifically, writing programs as simple equations means that reasoning about those programs follows the exact same patterns and tools that students learned in high school algebra (and that we reinforce in the early parts of the course.) And by carefully choosing the programs that we ask students to write and the properties that we ask them to verify, we can ensure that the proofs can be found by the theorem prover ACL2 with only a minimum of assistance from the students.

In the end, students learn much about software, how to build it and how to think about it. The majority of students will not take any further courses in computer science, but they will take with them a new way of thinking about computation to their diverse disciplines. And those students who do end up pursuing advanced courses in computer science will have a bird's eye view of the field that we hope will stay with them and give them a unique perspective as they study the engineering details that are needed to construct large-scale computing systems.

5. ACKNOWLEDGMENTS

The authors would like to thank the National Science Foundation for supporting the work described in this paper under Grant No. 1016532.

6. REFERENCES

[1] Hal Abelson, Ken Ledeen, and Harry Lewis. *Blown to Bits: Your Life, Liberty, and Happiness After the Digital Explosion.* Addison-Wesley Professional, 2008.

[2] Harsh Chamarthi, Peter Dillinger, Matt Kaufmann, and Panagiotis Manolios. Interactive testing and interactive theorem proving. In *Proceedings of the Tenth International Workshop of the ACL2 Theorem Prover and its Applications (ACL2-2011)*, 2011.

[3] Stephen Cooper, Wanda Dann, and Randy Pausch. Teaching objects-first in introductory computer science. *SIGCSE Bull.*, 35(1):191–195, Jan. 2003.

[4] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[5] Carl Eastlund. DoubleCheck your theorems. In *Proceedings of the Eighth International Workshop on the ACL2 Theorem Prover and its Applications (ACL2-09)*, 2009.

[6] Carl Eastlund, Dale Vaillancourt, and Matthias Felleisen. ACL2 for freshmen: First experiences. In *Proceedings of the Seventh International Workshop of the ACL2 Theorem Prover and its Applications (ACL2-2007)*, 2007.

[7] Matthias Felleisen. The first year. `http://www.ccs.neu.edu/home/matthias/Presentations/FirstYear/first%20year.pdf`.

[8] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. *How to Design Programs: An Introduction to Programming and Computing.* MIT Press, Boston, 2001.

[9] Brian Harvey and Jens Mönig. Bringing "no ceiling" to scratch: Can one language serve kids and computer scientists? In *Constructionism 2010 (Paris)*, 2010.

[10] Yu-Chang Hsu, Kerry Rice, and Lisa Dawley. Empowering educators with google's android app inventor: An online workshop in mobile app design. *British Journal of Educational Technology*, 43(1):E1–E5, Jan. 2012.

[11] Joint Task Force on Computing Curricula. Computing curricula 2001 computer science. *Journal of Educational Resources in Computing (JERIC)*, 1(3es), Sept. 2001.

[12] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Press, 2000.

[13] Takayuki Kimura. Recursive programming in english for freshmen. *SIGCSE Bull.*, 9(1):129–132, Feb. 1977.

[14] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, Apr. 2010.

[15] The College Board. Computer science: Principles. `http://www.collegeboard.com/prod\_downloads/computerscience/ComputationalThinkingCS_Principles.pdf`, 2011.

[16] The College Board. CS principles pilot sites. http://www.csprinciples.org/home/pilot-sites, 2012.

[17] Franklyn Turbak, Constance Royden, Jennifer Stephan, and Jean Herbst. Teaching recursion before loops in CS1. *The Journal of Computing in Small Colleges*, 14(4):86–101, 1999.