How Computers Work: Computational Thinking for Everyone

Rex Page*
University of Oklahoma
Norman, OK, USA

Ruben Gamboa
University of Wyoming
Laramie, WY, USA

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? Assume they have the standard college prep background. This would include basic algebra, but not necessarily more advanced mathematics. They would have written a few term papers, but would not have written computer programs. They could surf and twitter, but could not exclusive-or and nand. What about computers would interest them or help them place their experience in context? This paper provides one possible answer to this question by discussing a course that has completed its second iteration. Grounded in classical logic, elucidated in digital circuits and computer software, it expands into areas such as CPU components and massive databases. The course has succeeded in garnering the enthusiastic attention of students with a broad range of interests, exercising their problem solving skills, and introducing them to computational thinking.

1 One and Done

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? Assume they have the standard college prep background. This would include basic algebra, but not necessarily more advanced mathematics. They would have written a few term papers, but would not have written computer programs. They could surf and twitter, but could not exclusive-or and nand. What about computers would interest them or help them place their experience in context?

This paper discusses one of the many possible answers to this question. It describes experiences in teaching an honors course for students from a variety of disciplines at the University of Oklahoma. The students have varied interests and come from all college levels, first year to fourth year. They can choose from many courses to satisfy their honors requirements, from Beatles History to Moby Dick to What is Science? This course, called "How Computers Work: Logic in Action," has succeeded in getting their enthusiastic attention and exercising their problem solving skills.

The course includes some computer programming, but does not dwell on it. Students get enough experience to know what software is, but not enough to take on serious software development projects. The material helps students understand what makes automated computation possible by expanding on computational principles and overarching insights rather than the details needed in the practice of engineering. Most of the students will not continue with additional study in computer science. They will not become practicing engineers in hardware or software development.

Computers are demystified. Students grasp the fundamentals that make automated computation possible. Those that decide to go further have a better initial understanding of the big ideas than many of the students who decide early on to focus their studies in computing. It gives these students a leg up, but

*This material is based upon work supported by the National Science Foundation under Grant No. 1016532.

To appear in EPTCS.

does not overlap directly with mainstream material from a computer science or computer engineering program.

The big ideas in the course are

1. the correspondence between digital circuits and formulas in logic,
2. how abstractions facilitate combining solutions to small problems to form solutions to big ones,
3. how algebraic formulas can specify computations,
4. how models expressed in software capture the behavior of processes and devices,
5. how important, complex algorithms derive from simple, definitional properties,
6. how different definitional properties can produce the same results at vastly different computational expense,
7. how computational expense makes some useful devices feasible and renders others infeasible,
8. and how all of these ideas 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.

2 Demographics

"How Computers Work: Logic in Action" has been offered twice, so far, as part of a collection of "perspectives" courses in the Honors College at the University of Oklahoma. As one of many requirements for earning a degree with honors, students must complete two perspectives courses. The first offering of this perspectives course was in spring, 2011. By popular demand, a repeat offering took place in spring, 2012, and we expect that the course will be offered again in 2013.

Perspectives courses in the Honors College are limited to nineteen students. Nineteen students enrolled in the 2011 offering, but two dropped the course after a few weeks. The 2012 offering was oversubscribed at twenty students. One dropped, leaving a full contingent of 19 to complete the course.

Eight of the total of 36 students in the two offerings of the course had major fields of study outside science and engineering: history, letters, philosophy, linguistics, economics, drama, psychology, and business. Sixteen were majoring in science, eleven were engineering students, and one, a first-year student, had not yet decided on a major. One of the engineering students was in computer science, and three were in computer engineering. Engineering students scored, on the average, three percentage points higher on examinations than students in science and five points higher than students outside science and engineering.

Almost 80% of the students (28 of 36) were in their first two years of college. About 60% of the students had some prior experience in programming. In most cases this was a course in high-school or college, but five students had been programming for more than two years. None had any prior experience in functional programming. Students with prior programming experience scored, on the average, four percentage points higher on examinations than students without programming experience.

All were honors students, which means they would have held a grade-point average of at least 3.4 (out of a possible 4.0) at the time of joining the honors program and would need to maintain at least that average to have an honors designation ("cum laude") on their diploma at graduation. In addition, enrolling in honors courses suggests a greater-than-average level of self motivation. Honors students tend to be well-engaged in their studies, and they participate energetically in class. They ask interesting questions, and vague or sloppy answers seldom go unremarked.

To summarize, 22% of students enrolled in the course were majoring in humanities, social sciences, or business. About 31% were engineering majors (mechanical, electrical, chemical, computer engineering, computer science), and 44% were science majors (physics, chemistry, biochemistry, microbiology, meteorology). Two of the science students were simultaneously working on a degree in mathematics. About 78% of the students were in their first two years of college, 22% in the third or fourth year. Engineering students scored higher, on the average, on examinations than students in arts and sciences, but not by much. Among arts and science students, science students did marginally better, but the margin was only two percentage points.

3 Course Content

3.1 Equations

Much of the material in the course centers around equations. For example, software is introduced as operations that transform operands to results. Operations are described informally, and students are asked to specify tests that the operations would have to satisfy if they were functioning correctly. For example, imagine that `cons` is an operator that inserts a new element at the beginning of a list.

$$(\texttt{cons}\ x\ [x_1\ x_2\ \ldots\ x_n]) = [x\ x_1\ x_2\ \ldots\ x_n] \qquad \{cons\}$$

Informally, we use square brackets to delimit lists and n to denote an arbitrary, natural number. So, in the above example $[x_1\ x_2\ \ldots\ x_n]$ stands for a list with n elements, and $[x\ x_1\ x_2\ \ldots\ x_n]$ is a list whose first element is x and whose subsequent elements are x_1, x_2, and so on up to x_n.

We put labels for equations on the right in curly braces. Later, we refer to an equation by its label. We would refer to the foregoing equation as $\{cons\}$.

As dual operators to `cons`, we have `first`, an operator that extracts the first element of a list, and `rest`, which returns a list identical to its operand, but with the first element missing.

$$(\texttt{first}[x_1\ x_2\ \ldots\ x_{n+1}]) = x_1 \qquad \{fst\}$$
$$(\texttt{rest}[x_1\ x_2\ \ldots\ x_{n+1}]) = [x_2\ \ldots\ x_{n+1}] \qquad \{rst\}$$

Students have no difficulty convincing themselves that if `cons`, `first`, and `rest` fail to satisfy either of the following equations, $\{fst\text{-}id\}$ and $\{rst\text{-}id\}$, there must be something wrong with at least one of the operators.

$$(\texttt{first}(\texttt{cons}\ x\ [x_1\ x_2\ \ldots\ x_n])) = x \qquad \{fst\text{-}id\}$$
$$(\texttt{rest}(\texttt{cons}\ x\ [x_1\ x_2\ \ldots\ x_n])) = [x_1\ x_2\ \ldots\ x_n] \qquad \{rst\text{-}id\}$$

3.2 Tests

In tandem with these informal equations, we introduce a formal notation: the DoubleCheck testing framework of Dracula [2]. This makes it possible for students to practice expressing their expectations in a form that allows the computer system to perform tests automatically using random data.

```
(defproperty fst-id
  (x  :value (random-integer)
   xs :value (random-list-of (random-integer)))
  (equal (first (cons x xs))
         x))
(defproperty rst-id
```

```
(x  :value (random-integer)
 xs :value (random-list-of (random-integer)))
(equal (rest (cons x xs))
       xs))
```

Some students complain about oddities in the notation: "`equal`" for "=", prefix notation instead of infix, fully parenthesized formulas instead of relying on rules of precedence, etc. But, they accept these things as necessary for communicating with the Dracula computing system.

Complaints fade quickly. There are plenty of more challenging things for the students to think about. Besides, having multiple notations for the same mathematical object is a theme that comes up again when students see the correspondence between formulas in Boolean algebra and diagrams of digital circuits, with decimal numerals and binary numerals as representations of natural numbers, and in other examples.

3.3 Inductive Definitions

Inductive definitions appear painlessly in the context of the testing of expectations. An early example is an operator `append` for concatenating lists.

(`append` $[x_1\ x_2\ \ldots\ x_m]$ $[y_1\ y_2\ \ldots\ y_n]$) = $[x_1\ x_2\ \ldots\ x_m\ y_1\ y_2\ \ldots\ y_n]$

Students recognize that if the operator `append` failed to pass either of the following tests, $\{app1\}$ and $\{app0\}$, it could not be functioning properly.

(`append` (`cons` *x xs*) *ys*) = (`cons` *x* (`append` *xs ys*))	$\{app1\}$
(`append` [ ] *ys*) = *ys*	$\{app0\}$

Initially, we avoid viewing these equations as an inductive definition. They are billed as simple tests that a correctly functioning operator would pass. Later, we assert that any such collection of equations actually defines an operator, provided the equations have the following characteristics:

1. *Consistent*: no two equations specify different results for the same input;
2. *Comprehensive*: all forms of input must match the operands on the left side of at least one equation; and
3. *Constructive*: any inductive reference to an operator must comprise a reduced computation.

We describe an inductive reference to an operator, `op`, as an invocation of `op` on the right-hand side of an equation, $\{eqn\}$, that also refers to `op` on the left-hand side. The operands in the inductive reference must match the operands on the left-hand side of a non-inductive equation more closely than they match the operands on the left-hand side of $\{eqn\}$.

Students learn that under these conditions, all properties of the operator derive from the equations. In other words, the equations define the operator. We refer to these definitional characteristics as "the three C's." They are recurring theme from early on and throughout the course.

3.4 Inductive Proofs

The `append` operator is associative.

(`append` *xs* (`append` *ys zs*)) = (`append` (`append` *xs ys*) *zs*)

Students express this property formally in the notation of the Dracula DoubleCheck facility. They run the test and find that it succeeds.

```
(defproperty app-assoc
  (xs :value (random-list-of (random-integer))
   ys :value (random-list-of (random-integer))
   zs :value (random-list-of (random-integer)))
  (equal (append xs (append ys zs))
         (append (append xs ys) zs)))
```

Of course the associativity property, like all properties of the `append` operator, can be derived from its definitional properties, $\{app1\}$ and $\{app0\}$. We derive additional properties of functions mostly by substituting new, equivalent, formulas at strategic points to form new equations, a method entirely familiar from high-school algebra, except that the operators involved, instead of being addition, multiplication and the like, often deal with non-numeric data, such as lists. Since the definitional equations are usually inductive, most of our derivations cite induction as a rule of inference at some point in moving from one formula to the next.

A pencil-and-paper proof of the associativity property from the informal equations could be carried out as an induction on the length of *xs*. The base case, when *xs* is the empty list, cites the $\{app0\}$ equation twice.

$$\begin{array}{lll} & (\texttt{append}\ [\,]\ (\texttt{append}\ ys\ zs)) & \\ = & (\texttt{append}\ ys\ zs)) & \{app0\} \\ = & (\texttt{append}\ (\texttt{append}\ [\,]\ ys)\ \ zs) & \{app0\} \end{array}$$

In the inductive case, the length of *xs* is non-zero. That is, $xs = [x_1\ x_2\ \ldots\ x_{n+1}]$ for some natural number n. So, the inductive case can be argued as follows.

$$\begin{array}{lll} & (\texttt{append}\ [x_1\ x_2\ \ldots\ x_{n+1}]\ (\texttt{append}\ ys\ zs)) & \\ = & (\texttt{append}\ (\texttt{cons}\ x_1\ [x_2\ \ldots\ x_{n+1}])\ (\texttt{append}\ ys\ zs)) & \{cons\} \\ = & (\texttt{cons}\ x_1\ (\texttt{append}\ [x_2\ \ldots\ x_{n+1}]\ (\texttt{append}\ ys\ zs))) & \{app1\} \\ = & (\texttt{cons}\ x_1\ (\texttt{append}\ (\texttt{append}\ [x_2\ \ldots\ x_{n+1}]\ ys)\ zs)) & \{ind\ hyp\} \\ = & (\texttt{append}\ (\texttt{cons}\ x_1\ (\texttt{append}\ [x_2\ \ldots\ x_{n+1}]\ ys)\ zs)) & \{app1\} \\ = & (\texttt{append}\ (\texttt{append}\ (\texttt{cons}\ x_1\ [x_2\ \ldots\ x_{n+1}])\ ys)\ zs) & \{app1\} \\ = & (\texttt{append}\ (\texttt{append}\ [x_1\ x_2\ \ldots\ x_{n+1}]\ ys)\ zs) & \{cons\} \end{array}$$

The DoubleCheck property `app-assoc` is a formal statement of associativity, and students can direct Dracula to submit the property to the ACL2 theorem prover [3] for formal, mechanized verification. In this way, students gain experience with inductive definitions and with expressing expectations both formally and informally. The also learn to do informal, paper and pencil proofs and to use a mechanized logic to produce formal proofs.

3.5 Programming

Suppose a student wants to define an operator that extracts the first n elements from a list *xs*. Using the principle of the three C's described in Section 3.3, the student looks for equations that express properties of the operator that are consistent, comprehensive, and computational. Students who manage to conjure up the following equations have succeeded in writing a program for the `prefix` operator.

$$\begin{array}{ll} (\texttt{prefix}\ 0\ xs) = [\,] & \{pfx0\} \\ (\texttt{prefix}\ n\ [\,]) = [\,] & \{pfx-\} \\ (\texttt{prefix}\ (n+1)\ (\texttt{cons}\ x\ xs)) = (\texttt{cons}\ x\ (\texttt{prefix}\ n\ xs)) & \{pfx1\} \end{array}$$

We do not pretend that inventing these equations is easy. It calls for creativity and insight, and those things come from practice. Students need to do lots of exercises to learn the material. Gradually, they

build up their expertise, and along with it comes the ability to derive new properties of operators from definitional ones. The entire mechanism is built on ordinary, algebraic equations and classical logic.

To run their program for the `prefix` operator, they must formalize it. The mechanism for that is ACL2 [3]. Since there are three equations, the definition will need to say which formula for (`prefix` *n xs*) applies in what circumstances. It can use the "`if`" operator to make the appropriate selection. In this example, even though there are three equations, there are only two distinct formulas for the value of (`prefix` *n xs*) because the empty list is the result that `prefix` delivers in two of the equations. So, the "`if`" operator only needs to choose between two formulas, and the definition could be written as follows.

```
(defun prefix (n xs)
  (if (and (consp xs) (not (zp n)))
      (cons (first xs) (prefix (- n 1) (rest xs)))  ; {pfx1}
      nil))                                         ; {pfx0}
```

At this point, students can think about other properties of the `prefix` operator and test their expectations with DoubleCheck. One property they might think of is a relationship between `prefix` and `append`.

$$(\texttt{prefix}\ (\texttt{len}\ xs)\ (\texttt{append}\ xs\ ys)) = xs \qquad \{app\text{-}pfx\}$$

A formal, DoubleCheck definition of this property would mechanize the test.

```
(defproperty app-pfx ; preliminary version
  (xs :value (random-list-of (random-integer))
   ys :value (random-list-of (random-integer)))
  (equal (prefix (len xs) (append xs ys))
         xs))
```

All of the random tests that DoubleCheck generates pass, so the next step is a paper-and-pencil proof that the property holds for all lists. That goes through, too, by induction on the length of *xs*.

Unfortunately, ACL2 fails to complete a formal proof of the theorem corresponding to the `app-pfx` property. The problem is that, while the property holds for all of the random tests that DoubleCheck generates, it does not hold under all circumstances. If *xs* is not a list, but is, instead, some other kind of object, the property fails. To be an ACL2 theorem, the propoerty must constrain *xs* to the domain of lists.

```
(defproperty app-pfx ; provable version
  (xs :value (random-list-of (random-integer))
   ys :value (random-list-of (random-integer)))
  (implies (true-listp xs)
           (equal (prefix (len xs) (append xs ys))
                  xs)))
```

Even then, ACL2 does not succeed with a proof until it imports the standard theorems of numeric algebra, which have been derived in a certified package distributed with the ACL2 system. So, in this example, students learn to deal with a few of the complications that can arise when moving from an informal environment to a formal one.

Programming examples in the course progress from simple ones like `prefix` to complex ones like merge-sort and AVL trees. All of them include testing and deriving, informally and formally, additional properties from the definitional ones.

$x \vee False = x$	$\{\vee$ identity$\}$
$x \vee True = True$	$\{\vee$ null$\}$
$x \vee y = y \vee x$	$\{\vee$ commutative$\}$
$x \vee (y \vee z) = (x \vee y) \vee z$	$\{\vee$ associative$\}$
$x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$	$\{\vee$ distributive$\}$
$x \rightarrow y = (\neg x) \vee y$	{implication}
$\neg(x \vee y) = (\neg x) \wedge (\neg y)$	$\{\vee$ DeMorgan$\}$
$x \vee x = x$	$\{\vee$ idempotent$\}$
$x \rightarrow x = True$	{self-implication}
$\neg(\neg x) = x$	{double negation}

Figure 1: Basic Boolean equations (axioms)

	$(x \vee y) \wedge y$	
$=$	$(x \vee y) \wedge (y \vee False)$	$\{\vee$ identity$\}$
$=$	$(y \vee x) \wedge (y \vee False)$	$\{\vee$ commutative$\}$
$=$	$y \vee (x \wedge False)$	$\{\vee$ distrubutive$\}$
$=$	$y \vee False$	$\{\wedge$ null$\}$
$=$	y	$\{\vee$ identity$\}$

Figure 2: $\{\wedge$ absorption$\}$: $(x \vee y) \wedge y = y$

3.6 Propositional Logic and Digital Circuits

Equations provide a basic theme that permeates the course. Because the equations of Boolean algebra are so much like those of numeric algebra, which students are familiar with, the technical part of the course starts in this domain.

Propositional logic is derived from ten basic equations of Boolean algebra (Figure 1). The traditional truth tables, absorption equations, and so on are derived by reasoning from the basic equations in the standard, algebraic way.

This gives students practice, early on, in the syntax matching and step-by-step reasoning that is used throughout the course, but in a tightly prescribed context, where keeping track of formulas is relatively simple. Figure 2 displays a typical proof that students would see or be asked to derive from the basic equations, or from other equations such as $\{\wedge$ null$\}$ derived earlier from the basics.

After some study of propositional logic, digital circuits are introduced as an alternate notation for Boolean formulas. We focus on and-, or-, and not-gates, but we also draw circuit diagrams with exclusive-or, nand, nor, and other standard gate symbols. We show, by reasoning from the basic Boolean equations, how the behavior of any circuit can be fully realized in terms of nand-gates alone. As an exercise, students show that an implication gate is universal in the same sense as nand.

Gradually we work up to a ripple-carry adder circuit. All of the algebraic support for twos-complement arithmetic is defined and justified using standard equations of numeric algebra and, formally, by defining ACL2 operators to carry out arithmetic on binary numerals. In this way, the students see a model in software of a ripple-carry adder, along with a diagram of a digital circuit for the adder at the gate level. The fact that the operations the circuit performs are consistent with ordinary arithmetic on numbers is proved by induction, both informally (paper and pencil) and formally through ACL2. In addition, a software

model for bignum addition and multiplication on binary numerals is developed, and its operations are justified by informal and formal, mechanized proofs.

This helps students understand how physical devices can perform computations, and that provides a basis for understanding how computers work at the circuit level. Students acquire an understanding of how circuits do what they do, and how engineers can know, for certain, some of the operational properties of circuits.

3.7 Massive-Scale Computing: Websites and User-Provided Content

A course emphasizing the logical foundations of computing can leave students (computing majors, especially, but other students, too) with the erroneous impression that none of this is relevant for real-world computing. To address this situation, students are presented with a selection of real-world applications that they are familiar with. The applications are chosen carefully so that they resonate with students and they elaborate on principles that have already been discussed in class.

One such application is Facebook. Students see a quick overview of the history of web applications, with a focus on the distinction between Web 1.0 and Web 2.0 applications. What students learn is that Web 2.0 applications mix content from application producers and with content from consumers. Different consumers may see completely different results, due to customization and personalization. Facebook, the ultimate Web 2.0 application, actually provides very little content in the traditional sense. Most of the content is contributed by each user's circle of friends.

Students also learn the concept (but not the working details) of relational databases and how they can be used to generate content for Web 1.0 applications, such as traditional storefronts. Then students learn the Achilles Heel of relational databases, namely that join operations on large tables are prohibitively expensive. As a case in point, the join of the "friends" and "statuses" tables that could, in principle, support Facebook is infeasible.

So how does Facebook do it? Facebook developers solved this problem by building their own non-relational database called Cassandra [4]. At a high level, Cassandra acts like a simple key-value store, and the students have experience with this concept, having earlier studied data structures such as search trees and AVL trees. But Cassandra is based on the concepts of consistent hashing, database sharding, data replication, and eventual consistency. Although the details of these features are very technical, at a high level of abstractions they are easily accessible, even to non-technical students. For example, students can grasp how the Cassandra ring architecture enables both the massive throughput required by Facebook and reliability in case of system failure, which is certain to occur from time to time in such a large application.

This discussion does not prepare students to understand how NoSQL databases such as Cassandra are implemented, or even how to use them. Rather, it serves to convince students that ideas and techniques that they have already seen can be scaled up to build large, important applications such as Facebook.

3.8 Massive-Scale Computing: Web Search Engines

Another example, Google's technology stack, reinforces the lesson that the basic computer principles students have studied in the course have real-world implications. The students are, of course, familiar with Google products, such as the search engine and Gmail, as paragons of web applications. Now we expose them to a piece of Google's technology—MapReduce [1].

MapReduce is part of Google's approach to distributed computing. Jobs are broken down into map and reduce steps that operate on dictionaries (key/value data structures). A MapReduce program can be

developed entirely on a single computer. Then, the MapReduce framework takes care of the details of executing individual map and reduce tasks on hundreds or thousands of computers.

This particular technology is chosen because it is easily motivated by Google's huge scaling needs. In this way students see, once more, how dealing with large scale is the main difference between computing concepts as they have experienced them and engineering as practiced in real-world computing. Another reason for choosing MapReduce is because of its roots in functional programming, which immediately ties into programming concepts that the students have learned in the course.

Students see some MapReduce operations actions expressed in the form of ACL2 functions. Examples include distributed word count and distributed grep. Students also see how the MapReduce framework can be used to perform meaningful work at scale, such as inverting the graph of internet links, which is then used to calculate PageRank [5] for Google's search results.

4 Where Do We Go from Here and Why?

A course including topics in classical logic, digital circuits, programming, testing, verification, and other major computing concepts, all of it couched in terms of a familiar form of reasoning, namely algebraic equations, can provide a comfortable, yet challenging environment for interested students, regardless of background, assuming a standard, college prep education, including especially high-school algebra. Such a course can provide a basis for understanding in a fundamental way what computers do and how they do it. It is one way to introduce students to computational thinking.

This is not a "soft-skills" course. It calls for careful thinking, and it rewards studious attention. Yet, it is accessible and interesting to a broad base of college students. This combination of depth, challenge, and reward offers students something new and valuable.

An essential component of this enterprise is an equation-based programming language with a property-based testing facility. A mechanized logic with a quick-entry learning curve enhances the experience and the educational impact relative to the learning effort that students invest. A conventional programming language will not serve because it cannot be understood in terms of classical logic and the standard mechanisms of algebraic reasoning. The theme of equations cannot be carried throughout a course in which the programming component relies on an imperative paradigm.

We are writing a textbook that takes the approach and includes the material discussed in this paper. Drafts of the text have been used to provide readings in the course. We plan to develop interactive, web-accessible learning tools, including mini-tutorials, exercises, and automated assessment of solutions for instant feedback. Lecture notes, homework projects, and examinations are available upon request to educators who want to incorporate some of these ideas in their work.

The College Board has proposed a new course in the principles of computer science designed for college preparatory students and first-year college students [6]. It emphasizes computational thinking. The proposal elaborates seven big ideas and key concepts: creativity, abstraction, data, algorithms, programming, internet, and impact. Based on the descriptions in their proposal, we believe that the material in our course and its accompanying text provide an effective learning environment for those ideas and concepts. A course following this approach would be one way to introduce computational thinking to a broad range of students.

References

[1] J. Dean & S. Ghemawat (2004): *MapReduce: Simplified Data Processing on Large Clusters*. In *Proceedings of the* 6^{th} *Symposium on Operating System Design and Implementation (OSDI'04)*. p. 10.

[2] C. Eastlund (2009): *DoubleCheck Your Theorems*. In S. Ray & D. Russinoff, editors: *Proceedings of the* 8^{th} *International Workshop on the ACL2 Theorem Prover and its Applications*, pp. 42–46.

[3] M. Kaufmann, P. Manolios, & J.S. Moore (2000): *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers

[4] A. Lakshman & P. Malik (2010): *Cassandra: A Decentralized Structured Storage System*. *ACM SIGOPS Operating Systems Review* 44(2), pp. 35–40.

[5] L. Page (2007): *Scoring Documents in a Linked Database*. U.S. Patent 7,269,587.

[6] The College Board (2011): *Computer Science: Principles*. Available at `http://www.collegeboard.com/prod_downloads/computerscience/ComputationalThinkingCS_Principles.pdf`.