Initial Experience with a Computational Thinking Course for Computer Science Students

Dennis Kafura, Deborah Tatar
Department of Computer Science
Virginia Tech
{kafura, tatar}@cs.vt.edu

ABSTRACT
Experience with the first offering of a computational thinking course for computer science (CT4CS) students is reported. The course is grounded in student interaction with fundamental, recurring concepts suggested by comparison with two sets of computer science principles. By using specialized, freely available tools and physical simulations it is possible to provide concrete, tangible learning experiences that neither require knowledge of nor the overhead of programming. Student end-of-term reflections indicate that the course deepened and broadened their understanding of computer science even when they had previously encountered a topic, and improved their computer science vocabulary.

Categories and Subject Descriptors
K.3.2 [**Computers and Education**]: Computer and Information Science Education

General Terms
Design, Experimentation

Keywords
Computational thinking, curriculum,CS0, pedagogy

1. INTRODUCTION
Like many departments across the country, the Department of Computer Science at Virginia Tech has worked to revamp the entry points to its undergraduate curriculum. A particular focus of concern has been the widespread perception that computer science is "just details of programming." This effort has included restructuring the sequence of programming courses, introducing an optional media computation first course, and adding an early required non-programming course in problem-solving.

Another, as yet experimental, approach is a "computational thinking" first course referred to as CT4CS. The name CT4CS is used because the conception of this course draws on the general computational thinking concern for promoting an early and deeper understanding of critical notions of computation [7, 10, 20, 21]. But this project differs from other efforts because it focuses on students interested in computer science as a likely area of study. Thus, rather than developing computational thinking for and embedded in other disciplines, we are interested in computational thinking as a core component and attractive aspect of computer science itself. Courses in computational thinking that target other fields of study (science, engineering, liberal arts, etc.) are clearly needed and well motivated. While the goal of computational thinking to help students develop "ways to think like a computer scientist" [20] is universally valuable, this goal is especially critical to students learning to become practicing computer scientists.

The overarching goal of this course is to create learning experiences that allow prospective computer science majors to encounter recurring, fundamental concepts in computer science early in their study. Through these experiences students explicitly learn a number of critical principles but, perhaps more importantly, develop a cognitive model for and underlying intuitions and mental models about computational phenomenon. In this way, students develop an initial, albeit incomplete, framework for their future study of computer science, a framework that aids them in recognizing connections across courses which may otherwise go unnoticed. From a pedagogical standpoint, this approach builds on Bransford's notion of contrasting cases [1] which emphasizes that learning is aided by seeing the introduction of new ideas as a solution to a pre-existing problem. Students, therefore, are better positioned to recognize key ideas later on, integrate them into the particulars of more advanced courses, and enrich their epistemologies throughout the course of their studies. Instructors across the computer science curriculum can also assist students integrate and deepen their understanding by explicitly referencing the concepts. The course objectives were to:

- better inform prospective majors about the nature of computer science,
- engender a set of intuitions, perspectives, thought processes, or mental models that are indicative of how computer scientists view the world and conduct the practice of their work,
- introduce recurring, fundamental concepts and structures – ideas that appear, perhaps implicitly, in several different contexts in computer science, and
- develop a more sophisticated computer science epistemology, including key vocabulary.

These goals are, of course, to be understood in the context of a single semester introductory course.

SIGCSE'11, March 9–12, 2011, Dallas, Texas, USA.

A feature of CT4CS is that it does not require prior knowledge of and does not use programming. The decision to avoid programming was motivated by four considerations. First, to convincingly illustrate that computer science is "more than just programming," the course allows students to encounter basic principles of computation without reference to programming. Second, in the absence of programming, the students' full time and energy is focused more clearly on the concepts because these concepts are not confounded with the extraneous details of how these concepts are expressed in a given programming language. Third, avoiding programming helps to diminish the effect of the students' prior, usually programming experience on their opportunity to learn the concepts presented in the course. The course can deepen the knowledge of students with prior experience and simultaneously reach students without such background. Fourth, the strict avoidance of programming was adopted to ensure that the course design would explore the curriculum space as distinct from possible from programming approaches, including media computation style approaches.

Nonetheless, CT4CS allows *concrete and tangible* work to be done involving material drawn from every level of the curriculum. The work is described as concrete and tangible for three reasons. First, in some cases the students are able to *construct* artifacts or systems of interacting elements and explore their behavior. Students encounter not only the abstract idea but are able to undertake practical work that embodies and explores the idea. Second, in some cases the students are introduced to a specific encoding (notation) so that their thinking about an idea can be externalized, shared, and comment on. Third, in some cases the students were involved in conducting physical simulations that engaged them in manipulating physical objects representing aspects of a concept in material form. Examples of the concrete and tangible work included in the course are: using a finite-state machine simulator to develop a gene acceptor based on the genetic code presented in a junior level computational biology course, using a taxonomy modeling tool to represent the relationships of real-world examples that students would encounter in senior level courses in databases or information retrieval, using a Petri net simulator to study classical synchronization and protocol problems that students encounter in junior or senior level operating systems and networking courses, and using a lambda calculus evaluator to deal with issues of binding that are explicit in junior or senior level courses in programming languages or compilers and which are implicit in programming courses.

Section 2 provides a description of the course topics and tools. Section 3 compares CT4CS to other related approaches. Section 4 evaluates the course using student feedback from an end-of-term questionnaire. Section 5 draws some conclusions based on the evaluation and the comparison to other approaches.

2. COURSE DESCRIPTION

The course was offered as an alternative to an existing required introductory class. Students self-selected to register for the course which met over a 15-week semester in two 75-minute sessions each week. The room was outfitted with small round tables to accommodate in-class work by teams of two or three students. An overview of the course topics is given in Table 1. The time devoted to each topic is measured in weeks. One week consists of two 75-minute classes and the associated time students spend working on assignments and readings outside of the class meetings. Additional information can be found at: http://www.cs.vt.edu/~kafura/ComputationalThinking .

Broadly, the topics are divided into two categories were based on a definition of computer science that emphasizes the representation (modeling) and manipulation (via engineered systems) of information that is inherent to scientific, natural, social, and others kinds of systems. The modeling topics focus on how information is used to represent the attributes and relationships of real-world artifacts and systems. The engineering topics focus on issues related to the construction of computational systems.

2.1 Modeling Topics

The modeling topics focused on state and behavior, abstraction, and relationships. Finite state machine diagrams were used to explore the importance of stateful behavior, recognizing that in a given state the system is only capable of certain actions, that the observed behavior of a system arises from its transitions among its possible states, and that reactive behavior occurs when the system interacts with an external world that drives the system from one state to the next. A particular kind of state-oriented system is an acceptor. Using a tool (JFLAP [11]), the students were able to build and test acceptors for various kinds of structured data including one that performed a simple form of gene recognition (allowing a discussion of the relationship of computer science to biology). The graphical form of an acceptor was also related to a text-based form using BNF grammars. The abstraction topic studied how artifacts have a rich set of attributes, that representing an artifact depends on the point of view of the modeler, and that there are a variety of graphical and textual forms for representing the abstraction. A physical simulation engaged the students in the modeling of a book: identifying different modeler points of view and creating attributes that are relevant from each point of view. The relationships topic looked at very basic ideas of knowledge representation. Using Protégé [14], the students were able to define the attributes of a collection of entities (pizzas) and observe what inferences could be made automatically about the categorization of the entities.

2.2 Engineering Topics

The engineering part of the course consisted of five topics. The notion of concurrency and the problems of concurrency control were introduced by a physical simulation where the students acted out a scenario of updating a shared memory and observing the loss of consistency. The modeling of the correct synchronization for several problems (traffic lights, reader-writer) was done using a graphical tool (Snoopy [15]) for editing and executing Petri nets. Snoopy was also used to illustrate "layers of abstraction." A simple layered system was built for the AB protocol. Using the Petri net tool, the students modeled a noisy channel, the AB protocol layer, and a bi-directional channel layer. The last four engineering topics focused on concepts related to programming languages and software systems. First, the lambda calculus was used both to explore the ideas of binding and scope as well as give some insight into the theory of programming languages and computation. A graphical tool for composing and evaluating lambda expressions (LambdaTeacher [12]) was used. Second, testing of software systems was undertaken using two exercises.

Weeks	Topic		Tools/Methods
.5	Modeling	Definition of computer science	Guided discussion
2		State and Behavior	Applet; tool for creating and exercising an acceptor (JFLAP); BNF grammar; BNF visualizer
2		Abstraction: modeling/design perspective	Physical simulation; UML diagrams, Venn diagrams, XML, tree diagrams
1.5		Relationships	Tool (Protégé) for representing, inferring, and visualizing relationships; ontologies
1	Midterm exam and return/review		
1.5	Engineering	Concurrency	Physical simulation; tool (Snoopy) to create and simulate Petri nets
1		Abstraction: engineering perspective	Layered systems; tool (Snoopy) to build layered protocol
2		Binding, scope, theory: lambda calculus	Tool (Lambda Teacher) for editing and evaluating lambda expressions
1		Testing	Testing applet for simple (triangle classification) problem; Exercise (Web-CAT) for more complex (sentence parsing) problem
1		Debugging	Puzzle (Sudoku) tool with history and backtracking
1		Memory, pointers, data structures	Physical simulation; C syntax
.5	Review, Course evaluation		

Table 1: CT4CS Course Overview

The first exercise allowed the students to enter test cases for a program to classify triangles (as equilateral, isosceles, or scalene). The tool used provided code coverage statistics and error detection percentages against a hidden collection of correctly and incorrectly programmed solutions. The second exercise involved the development of test cases for a grammar-based sentence parser. An automated system (WebCAT [19]) developed at Virginia Tech was used to run the test cases against a model solution and provide code coverage feedback. The third topic, debugging, explored unwinding a sequence of decisions to find a flawed decision that led to an erroneous state. A system for solving Sudoku puzzles [18] was used to present the students with an incorrectly solved puzzle. The tool provided a history and navigation controls so that students could "debug" the puzzle's incorrect solution. The last of the four topics was data structures. A physical simulation of a linear computer-like memory was conducted. The students were engaged in finding ways to map a tree structure (built of index cards and strings) into a linear memory. Notions of representing relationships via adjacency in memory and pointers arose from this simulation. The syntax of C was used to show how these ideas could be represented in a programming language.

3. COMPARISON

The CT4CS course can be compared with other computational thinking courses, definitions of the principles of computation, and repositories that provide resources for teaching topics in computer science.

CT4CS differs from other computational thinking approaches in one or both of two ways. First, some computational thinking courses are focused on students majoring in fields other than computer science (e.g, [8] focusing on science majors). These courses are valuable and seek to portray computation through topics and terms meaningful to other fields. CT4CS focuses exclusively on computer science majors and the terms of reference and topics that have the most impact on their future learning. Second, many courses use some form of programming either in lexical (e.g., [6]) or graphical (e.g., [17]) form. CT4CS explicitly avoids the use of programming languages so that the students can be engaged in topics that might be difficult to approach via programming. Admittedly the absence of a programming capability also limits some topics that can be approached in CT4CS. However, this is a tradeoff that seems reasonable given the topics that are addressed and the fact that the students will have multiple subsequent chances to encounter other important computing ideas. The specific computational thinking courses cited above are only representatives of many other similar courses from which CT4CS is differentiated in the ways just noted.

In comparison to principles of computing, CT4CS embodies five of the seven Big Ideas defined in the College Board's CS Principles and employed five of their seven computational thinking practices [2]. Specifically, CS4CS shows that computing is a creative human activity (Principle 1) by the creation of tangible artifacts, involved the use of abstraction to reduce information and detail (Principle 2), demonstrated knowledge creation (Principle 3) via ontologies, examined systems and networks (Principle 6) via concurrency and protocols, and illustrated how computing enables innovation in other fields via the use of examples from, for example, biology in the gene recognizer. The course used abstraction and models (Practice 3), analyzed problems and artifacts (Practice 4), connected computation with other fields (Practice 6), and engaged students in teamwork (Practice 7). Additionally, the course involved the creation of computation artifacts (Practice

2). While *programming* artifacts were not created, numerous other *computation* artifacts were created among which were test cases, UML diagrams, XML encodings, data structure declarations, BNF grammars, Petri nets, acceptors, lambda expressions, and an ontology. We believe that these kinds of artifacts deserve to be considered on an equal par with programming artifacts for several reasons. First, doing otherwise devalues the real work that computer scientists invest in the creation and use of these artifacts. Second, we are trying to counter the stereotype that limits computer science to programming. This stereotype is reinforced when we value only programming artifacts. Furthermore, we often deny students' early exposure to the very elements that make computer science of enduring interest during the bulk of most computer scientists' careers. Last, our experience in the course indicates that these non-programming artifacts enable student learning.

Another set of principles is Denning's framework [4], one axis of which is a set of computing practices and the other axis of which is a set of principles for design and mechanics. The topics in the CT4CS course have strong overlap with the elements of this framework. Specifically, the coordination and communication elements of mechanics resonate with the concurrency and engineering abstractions in the CT4CS class; the automation element is reflected in the relationships topic where automatic deduction of relationships is seen; to a lesser extent the computation element is touched on by the lambda calculus and the recollection element by the topic on data structures. The relationship of the CT4CS topics to the design aspects was centered on the simplicity element, showing how abstraction is used to simplify the description of artifacts. The practices axis contained two elements that related to topics in CT4CS. Many of the topics in the "engineering" half of the course relate well to the "engineering systems" practice. Similarly, many of the topics in the "modeling" half of the course relate well to the "modeling and validation" practice.

Finally, there are repositories of materials for teaching computing, perhaps the best exemplar of which is the Computer Science Unplugged collection [3]. CT4CS shares with the "unplugged" materials the goal of providing tangible and concrete experience about computing concepts without programming. A important difference between the two is the academic level of the audience which dramatically influences the level of sophistication of the topics and the presentation of these topics. Nonetheless, there might be contributions that CT4CS can make to the unplugged repository and there may be materials or ideas that can be adapted from the repository for use in future offerings of CT4CS.

4. EVALUATION

The observations reported in this section are based on the students' answers to a questionnaire. All of the 15 students, three of whom were female, were in their second semester at Virginia Tech. The questionnaire was administered on the last regularly scheduled class meeting. The students were told in advance of the general nature of the questionnaire (though not the specific questions). The questionnaire was distributed in electronic form and completed anonymously by students using their personal laptops. The students were not provided lists of course objectives or the course topics. The students had the full class period (75 minutes) to complete the questionnaire along with two other short standard course evaluations. All of the students completed the questionnaire before the end of the class. The questionnaire was divided into two parts: a student profile and student reflections. The student profile consisted of five questions about the student's background and interests in computer science. The student reflections contained eleven free response questions about various aspects of the course. The number of students in the class is not sufficient for statistical analysis but qualitative analysis of the results and the student reflections offers insights into student experiences. Furthermore, the argument for this approach lies only partially in precisely what was learned during the semester, and more fully in the way it is thought to smooth the path towards the future. However, the reflections offer important insight into the students' experience, and if their experiences are insufficiently good, then the long-term benefits of the approach are moot.

The students as a whole had substantial prior experience in computer science. Two-thirds of the class had programming experience of at least one semester and almost half of the class (7 of 15) had at least a full year of experience. This experience was typically described as an AP CS course using Java. There were only two students who had neither high school CS experience or completed a CS class at VT. There was only one student who had no prior CS coursework (in high school or at VT) and was not taking another CS class concurrently.

Observation 1: *The students reported that the course topics deepened their knowledge and perspective on computer science.* A third of the students (5 of 15) indicated that the course had a deep impact on their appreciation for or approach to computer science. This was expressed variously as gaining insight into the "why" behind programming, better understanding of what it means to be a computer scientist, or enabling future problem-solving with a deeper understanding of computing. Another group (3 of 15) reported more generically that they learned new concepts related to computing. A third group (3 of 15) reported that the course reaffirmed or clarified concepts that they already understood.

Observation 2: *The students reported that the course offered a number of new (to them) concepts and/or improved their understanding of concepts they had already seen.* Eight specific ideas were mentioned as being new. Four of these ideas (grammars/acceptors, finite state machines, Petri nets and concurrency, lambda calculus) were identified in this way by over half of the class (at least 8 out of 15). A minority (5 out of 15) identified two other topics (abstraction, ontologies) in this way. Two other topics were identified as new to the students by smaller groups (testing – 3 out of 15, debugging 2 out of 15). A strong majority (10 of 15) of the class reported that the class had improved their understanding of a concept that they had previously encountered. Students (6 out of 15) also reported that they had previously encountered the concepts of abstraction, generalization, or modeling but found that the course improved their understanding of these concepts.

These responses are meaningful in two ways. First, the identified concepts are fundamental to computing and, thus, progress in deepening students' understanding of these concepts is meaningful. Second, despite the high overall level of experience, the students found the course presented a number of new ideas or deepened their appreciation of ideas they had already

encountered. Thus, the course content seemed to "stretch" and/or deepen the students' understanding of computer science. For example, one student reported that the course provided a "deeper look into not only *how* to model things, but *why* we model them that way" (italics in student's comment). A smaller group (3 of 15) reported similar gains in understanding of concurrency issues and another group (3 of 15) cited improvement in their understanding of testing/debugging. There was a small amount of overlap among these three groups (due to students citing more than one topic area); in total the three groups above accounted for two-thirds of the class (10 of 15).

Observation 3: *The students reported that a number of course topics strongly engaged their interes*t. Overall, there were 22 reasons given for why a particular idea was cited by a student as being of interest. The number or reasons is more than the number of students because some students citied more than one idea as being interesting.

A majority of the students (8 of 15) reported that concurrency was the most interesting idea they encountered in the class. The reasons for their choice of this idea varied and included: utility (dealing with real life or practical situations), challenging, involved logic or problem solving, novelty (something not thought about before)**.** Another group (4 of 15) identified the ideas of grammars and finite state machines as among the most interesting. The given reasons behind their choice included logic and problem-solving, and utility (dealing with real life or practical situations). A smaller group (3 of 15) identified data structures as the most interesting topic citing utility (2 of 3) and novelty as their reasons. These same reasons were cited by a similarly sized group (3 of 15) for the ideas related to testing/debugging.

These reasons can be categorized as utility (can be applied to real-life problems, will be useful in future work), logic/problem-solving (can be approached in a logical fashion, requires structured thinking, engages problem-solving skills), challenge (offered a mental challenge to think about), and novel (the idea or some aspect of the idea was new, a new way of thinking about the idea). The distribution of the reasons suggests that the students found interesting ideas that they could see as having high utility (cited 9 of 22 times), novelty (6 of 22), logic-problem solving (5 of 22), or challenging (2 of 22). While these results had a high degree of overlap, their diversity is very important. There is no single reason that a person should be interested in computer science. To the contrary, the goals of this class and of other changes to our computer science curriculum, is to bring people with a wide scope and perspective as well as deep engineering capabilities into computer science.

Furthermore, it is interesting to note that a single topic (e.g., concurrency) may appeal to different students for different reasons, making the task of designing an appropriate set of topics more promising -- and more challenging. The student feedback also gave good guidelines for determining how to improve the presentation of current topics or what aspects to look for in new topics.

Observation 4: *The students reported that the course helped them develop a better vocabulary for explain computer science issues.* A strong majority of students (12 of 15) reported that the course had improved their technical vocabulary. The degree of improvement was variously described, ranging from "definitely" to "somewhat". Two students did not believe that their vocabulary was improved and one student was not sure. This spread in reactions may be due to the fact that the word "vocabulary" means different things to different people. Vocabulary is an interesting metric because it represents utility both for itself and also as an indicator of the students' epistemology, that is, their notions about the interconnected web of ideas inherent in computer science, some of which are well captured in particular specialized words or phrases.

Observation 5: *The students reported that the physical simulations were useful to their learning.* As described in Section 2, the course pedagogy included three physical simulations illustrating ideas related to abstraction, concurrency, and data structures. The students unanimously believed that the physical simulations were useful. Eighty percent (12 of 15) of the students gave unqualified support for the value of the physical simulations. Twenty percent (3 of 15) noted that the simulations took too much time to conduct or were seen as being useful only to those who had not yet taken a CS class. This feedback is meaningful because it might be the case that (relatively) new university students would find the activity of a physical simulation not sufficiently sophisticated for a college-level course.

Observation 6: *The students expressed divided opinions on the ordering of this course with respect to an introductory programming course in computer science.* The single strongest opinion (7 of 15) was that this course should come before a programming course. Three of these students believed that this course would help form ways of thinking that would make a programming course easier. Two students believed that the ideas in this course were presented in a more abstract or less detailed way than these concepts would appear in a programming course so it was useful (or necessary) to take this course first. Two students gave no reason. However, others (4 of 15) indicated that there was equal value in taking this course before or after an introductory programming class. The one student who explained this choice pointed to the mutual reinforcing of concepts between this course and a programming course, believing that there was value in such reinforcement working in either direction. Two students believed that this course should be taken after a programming class. Both these students believed that the practical grounding achieved in a programming course was necessary before approaching these concepts in a more abstract way in this course. One student found it useful to be taking this class concurrently with a media computation programming class. One student offered no opinion.

While there was a near majority who favor taking this course before a programming course, the variety of opinion and reasons suggested that there is some latitude in where the course could be positioned in the overall curriculum. The students' reasons could be interpreted to mean that the choice depends on a student's learning style more than any prior study, or that the students, most of whom had taken programming before or concurrently, had difficulty imagining the counter-factual case. Pragmatically, other scheduling constraints (on the individual student or the institution) may also play a role in determining when the course is taken.

5. CONCLUSIONS

The experience with the initial course offering convinced us that a computational thinking course for computer science students was a viable concept. We found it possible to provide practical, concrete, learning experiences about a variety of important computing concepts using tools and physical simulation rather than programming. Student feedback suggests that the course fulfilled its specific course objectives to:

- better inform students about the nature of computer science (observation 1)
- to help develop intuitions and mental models (observation 2)
- present fundamental concepts as gauged by the comparison with two sets of principles of computing (see section 3)
- develop better epistemologies, as indicated by vocabulary (observation 4)

The student feedback also indicated that the physical simulations were very useful to student learning (observation 5), and that a course like this one could be flexibly positioned with respect to a programming course (observation 6).

Our own experience in teaching went beyond these specifics to suggest that students used the experience to encounter ideas in a way that would more firmly fix elements of computational thinking in their minds and render the treatment of these ideas in subsequent courses more tractable, more marked, and more profound.

We do not believe that the particular set of topics used in this first offering is the "right" set; but the experience seems to confirm the notion that a non-programming computational thinking course for computer science students is viable.

We are exploring the development of CT4CS in two ways. First the collection of topics presented in the course can be expanded. Other topics that we are investigating are in machine learning [16], natural language processing [13], ideas related to networks and behavior [5], and social networks [9]. Second, the examples used in class and in assignments can be enriched to better illustrate the connections of computation with other interesting problem domains similar to the use of gene recognition in conjunction with acceptors. We are interested in receiving suggestions and ideas on either of these improvements from the community.

6. ACKNOWLEDGMENTS

We would like to thank and acknowledge the contributions to the course of Denis Gracanin (finite state machine applet), Steve Edwards (testing applet and WEbCAT assignment), and Joon S. Lee (Sudoko system).

7. REFERENCES

[1] Bransford, J. D., Franks, J. J., Vye, N. J. and Sherwood, R. D. New approaches to instruction: Because wisdom can't be told. Cambridge University Press, City, 1989

[2] College Board, CS Principles, http://csprinciples.org/

[3] Computer Science Unplugged. See: http://csunplugged.org/

[4] Denning, P. J. 2003. Great principles of computing. *Commun. ACM* 46, 11 (Nov. 2003), 15-20. DOI= http://doi.acm.org/10.1145/948383.948400

[5] [NB] Easley, David and Kleinberg, Jon Networks, Crowds, and Markets, Cambridge University Press, 2010, a pre-publication draft is available at http://www.cs.cornell.edu/home/kleinber/networks-book/networks-book.pdf

[6] Freudenthal, E. A., Roy, M. K., Ogrey, A. N., Magoc, T., and Siegel, A. 2010. MPCT: media propelled computational thinking. In *Proceedings of SIGCSE'10* (Milwaukee, Wisconsin, USA, March 10 - 13, 2010). ACM, New York, NY, 37-41. DOI= http://doi.acm.org/10.1145/1734263.1734276

[7] Guzdial, M. 2008. Education: Paving the way for computational thinking. *Commun. ACM* 51, 8 (Aug. 2008), 25-27. DOI= http://doi.acm.org/10.1145/1378704.1378713

[8] Hambrusch, S., Hoffmann, C., Korb, J. T., Haugan, M., and Hosking, A. L. 2009. A multidisciplinary approach towards computational thinking for science majors. In *Proceedings of SIGCSE'09* (Chattanooga, TN, USA, March 04 - 07, 2009) ACM, New York, NY, 183-187. DOI= http://doi.acm.org/10.1145/1508865.1508931

[9] HarambeeNet: The SocialNets in Education Project, see: http://harambeenet.org/modules.html

[10] Henderson, P. B., Cortina, T. J., and Wing, J. M. 2007. Computational thinking. *SIGCSE Bull.* 39, 1 (Mar. 2007), 195-196. DOI= http://doi.acm.org/10.1145/1227504.1227378

[11] JFLAP, http://www.jflap.org/

[12] Mason, K. An Interactive Interpreter for Expressions in the Lambda Calucus, Honors Paper, Department of Computer Science, University of Adelaide, November, 1997.

[13] Natural Language Toolkit, see: http://www.nltk.org/

[14] Protégé, http://protege.stanfard.edu/

[15] C Rohr, W Marwan, M Heiner: Snoopy - a unifying Petri net framework to investigate biomolecular networks; Bioinformatics 2010 26(7): 974-975. http://www-dssz.informatik.tu-cottbus.de/index.html?/software/snoopy.html

[16] Russel, Ingrid and Markov, Zdravko, Machine Learning Experiences in Artificial Intelligence: A Multi-Institutional Project, see: http://uhaweb.hartford.edu/compsci/ccli/index.htm

[17] Ruthmann, A., Heines, J. M., Greher, G. R., Laidler, P., and Saulters, C. 2010. Teaching computational thinking through musical live coding in scratch. In *Proceedings of SIGCSE'10* (Milwaukee, Wisconsin, USA, March 10 - 13, 2010). ACM, New York, NY, 351-355. DOI= http://doi.acm.org/10.1145/1734263.1734384

[18] Sudoku player**:** http://poet.cs.vt.edu/tuple_games/

[19] WebCAT http://web-cat.org/WCWiki/WebCatWiki

[20] J. Wing, Viewpoint-Computational Thinking, *Commun. ACM* 49,3 (March 2006) 33-3.

[21] J. Wing, Computational thinking and thinking about computing, *Philosophical Transactions of the Royal Society*, A, 2008, 366, pp. 3717-3725. DOI=10.1098/rsta.2008.0118