Teaching Fundamentals of Computing Theory: A Constructivist Approach

Carlos Iván Chesñevar *
Department of Informatics – University of Lleida
Jaume II, 69 – 25001 Lleida, SPAIN
Email: cic@eup.udl.es

Ana Gabriela Maguitman
Computer Science Department – Indiana University
Bloomington, IN 47405-7104, USA
Email: anmaguit@cs.indiana.edu

María Paula González
Department of Informatics – University of Lleida
Jaume II, 69 – 25001 Lleida, SPAIN
Email: mpg@cs.uns.edu.ar

María Laura Cobo
Departamento de Cs. e Ing. de la Computación – Universidad Nacional del Sur
Av. Alem 1253 – B8000CPB Bahía Blanca, ARGENTINA
Email: lc@cs.uns.edu.ar

ABSTRACT

A Fundamentals of Computing Theory course involves different topics that are core to the Computer Science curricula and whose level of abstraction makes them difficult both to teach and to learn. Such difficulty stems from the complexity of the abstract notions involved and the required mathematical background. Surveys conducted among our students showed that many of them were applying some theoretical concepts mechanically rather than developing significant learning. This paper shows a number of didactic strategies that we introduced in the Fundamentals of Computing Theory curricula to cope with the above problem. The proposed strategies were based on a stronger use of technology and a constructivist approach. The final goal was to promote more significant learning of the course topics.

Keywords: Computer Science Education, Fundamentals of Computing Theory, Informatics Technology, Constructivism.

1. INTRODUCTION

A Fundamentals of Computing Theory course (FCT) involves different topics that are core to the Computer Science (CS) curricula and whose level of abstraction makes them difficult both to teach and to learn. Such difficulty stems from the complexity of the abstract notions involved and the required mathematical background. From our teaching experiences with second-year undergraduate FCT courses we found that many students do not feel as motivated and interested in mastering these topics as we expected. Their lack of motivation was due not only to the fact that they perceived the treatment of FCT topics as too 'biased' towards mathematics rather than computer science but also to the complexity of the topics themselves. As a result, those students were applying some theoretical notions somehow mechanically, with the consequence of later difficulties not only in the FCT course itself but also in the rest of the CS curricula.
In order to make the FCT course more motivating for our students we introduced a number of combined didactic strategies along the traditional curricula. Based on the work of [1, 3, 4, 5] we emphasized:

1. A contextualized teaching of different FCT notions.
2. Discovery learning processes based on the use of pedagogical software tools for FCT.
3. The use of the students' previous knowledge in the CS curricula.

*Research partially supported by project CICYT TIC2001-1577-C03-03 funded by the Ministerio de Ciencia y Tecnología (Spain).

4. The introduction of reinforcement and extension activities related to FCT topics.

The idea was not to simplify the contents of the course, but rather to enrich it by introducing a stronger use of technology and a constructivist approach. The final goal was to promote more significant learning of some central FCT topics emphasizing the fact that the course was introducing the mathematical foundations of several notions in CS rather than merely presenting abstract concepts of discrete mathematics.
This paper is structured as follows. First, in Section 2 we give an overview of the constructivist model and its application in the context of CS education. Then in Section 3 we describe four major strategies we have implemented and the results we have obtained. Finally, in Section 4 we present our conclusions.

2. CONSTRUCTIVISM MODEL AND CS EDUCATION

The theory of learning called constructivism claims that knowledge is actively constructed by the student, and not passively absorbed from lectures and textbooks [1, 8, 2]. According to this theory the construction of new knowledge is performed recursively, based on previous knowledge. The constructivist approach postulates that effective learning relies on an explicit process in which viable mental models are constructed. This process of construction and reconstruction of ideas is actively guided by the teacher. The ultimate goal of constructivism is to achieve significant learning [11], i.e. adequate mental models that will be available for use in different contexts.
There are several pedagogical proposals for the constructivist model. According to Bruner [3, 4, 5], learning is an active and explicit cognitive process in which learners construct new concepts based upon their current knowledge. The learner selects and transforms information, constructs hypotheses, and makes decisions relying on a cognitive structure that provides meaning to all his/her experiences. Bruner [3] postulates that the learning process is determined by four major aspects: a) the predisposition of the involved people, b) the way in which the knowledge is organized (internal and external coherence), c) the way in which knowledge is presented to the learner, and d) the learner capability to restructure knowledge. In more recent work [4, 5] the above ideas are expanded to capture the social and cultural nature of learning. Some of the main features of Bruner's constructivist approach can be summarized as follows:

1. The learning process must be related with experiences and contexts that are significant to the learner.

2. The learning process should be oriented towards a spiral curriculum organization to facilitate the learner reconstruction of knowledge.

3. The learning process must be designed to promote significant learning that allows learners to go beyond the information given.

In this context, teachers and students should engage in a dialogue where an active role of the student is constantly stimulated. One tasks of the teacher is to perform a didactic transposition, ie. to transform the information to be learned into a format appropriate to the learner's current state of understanding. Curriculum should be organized in a spiral manner, so that students continually build new knowledge upon what they have already learned.
Constructivism has been intensively studied by researchers in major teaching areas as pedagogy education [12] or mathematics education [8]. However, there is much less research related to the influence of constructivism in CS education. As pointed out in [2]:

> *"... Constructivism has been extremely influential in science and mathematics education, but much less so in computer science education [...] While many computer science educators have been influenced by constructivism, only recently has this been explicitly discussed in published work..."*

Following the constructivist principles we gradually introduced a number of combined teaching strategies along the traditional FCT curricula. Such strategies were expressed in a stronger use of the informatics technology and were tested during several semesters in a second-year, undergraduate FCT course. The main goal was to help students achieve significant learning. We discuss such strategies in the next section.

3. DIDACTICS STRATEGIES FOR SIGNIFICANT LEARNING IN FCT

Following [1, 3, 4, 5, 11] some of the major principles that lead to significant learning are the following:

1. New knowledge must be related with experience and social context significant to the learner. Our learning is intimately associated with our interaction with other human beings (teachers, peers, society, etc.). The social dimension of learning involves discovering the implications and human significance of what is being learnt.

2. New knowledge acquisition must encourage discovery and active learning. As the process of learning is a reconstruction of the student's mental models carried out by the student himself/herself, it demands an active attitude. By getting involved in his/her own learning process the student can perform meta cognition activities

as the result of thinking about the learning process itself. A student may be learning how to engage in a particular kind of enquiry (eg. scientific method), or how to become a self-directed learner. Discovery learning activities and the stimulation of meta cognition enable students to continue learning by themselves in the future with greater effectiveness.

3. The use of previous knowledge plays a major role in learning. Because the learning process is based on the construction and reconstruction of ideas, the learning process of highly abstract concepts is not possible without having some other structure developed from previous knowledge to built on. Therefore any effort to teach must be related to the mental models of the learner, providing a path into the taught material based on the learner's previous knowledge.

4. Including reinforcement and extension activities promotes significant learning. In order for the acquired knowledge to be significant it must be functional and operative, ie. applicable. Reinforcement and extension activities tend to achieve this goal. Application learning allows other kinds of learning to become useful.

Contextualizing FCT in Computer Science history

Including details of the historical evolution of the theory of computing along with the different topics of the FCT curricula has proved to be very helpful and motivating for our students. Although a FCT course is not centered on the history of computer science, it can be considerably enriched by introducing biographical notes, videos and articles associated with the historical context in which the theory of computing emerged as a new discipline. For example, when formalizing the notion of effective procedure and presenting the Turing-Church thesis, we devoted the initial part of those lectures to describing some relevant facts about the lives of Alan Turing and Alonzo Church. In this context students come to know, for example, that both Stephen Kleene and Alan Turing were two of the 31 students who did their PhD's under Church's direction, so that it becomes more clear why their research lines were pursuing similar goals.
Our experiences showed that the discussion of historical facts also helps students to identify how different theoretical concepts evolved through time, and how scientific research works and contributes to the growth of a new field of knowledge, as it was the case for the theory of computing between 1930 and 1950. Theory of computing history lends itself also very useful to make students aware of the importance of cross-breeding in research, as it shows how different knowledge areas such as mathematics, language theory and engineering met as computer science evolved through time. Students come across the fact that Turing, Kleene and Church's ideas in the 30's, Chomsky's grammar hierarchy in the 50's, Petri nets, as well as Shepardson and Sturgis' register machines in the 60's were different formalisms to give an answer to the same question of what computability is.
In order to integrate history of computer science with the topics presented in the different lectures, a number of links to biographical notes and other related material were added to the course webpage. In that way, those students visiting the course webpage to download lecture slides corresponding, for example, to the theory of Turing machines, were at the same time induced to contextualize that topic by following links to Alan Turing's life or the social and historical context of his work. As additional material to the course slides we also incorporated a "timeline" in which the evolution of the theory of computing was shown and contrasted with major historical facts (e.g. the motivations for Turing's work can be better understood in the context of the political situation of England during World War II).

Incentivating Active Learning with Theoretical Computer Simulators

In several opportunities students find it difficult to grasp underlying concepts within FCT because the abstract formal notation overwhelms them. In other areas of computer science (such as computer architecture), special software simulators provide an excellent teaching tool for enabling active learning of specialized knowledge through abstraction, interaction, and visualization [24]. In the area of theoretical computer science, this has led to the development of a number of so-called *theoretical computer simulators* as educational tools [10, 13, 19].
In a recent paper Chesñevar et.al [7] analyzed the main features of most theoretical computer simulators for teaching FCT, distinguishing two software categories, namely: (1) generic, multi-purpose software packages for teaching and integrating several related concepts of FCT and (2) software tools oriented towards simulating a specific class of automata with educational purposes (e.g. Petri nets, transducer automata, etc.). Most of such software tools are freely available via the Internet.
Theoretical computer simulators have proven to be an excellent and motivating link between theory and practice, encouraging active and discovery learning in our students. We introduced several simulation tools along with the different topics of the FCT curricula. There are multi-purpose simulator programs that proved to be a good choice as unified frameworks for simulating different kinds of automata. Below we briefly describe the main ones we have introduced.

- MINERVA [10] is an interactive and visual tool implemented in Java that allows the design, de-

bug and execution of different kinds of automata (FSA, PDA and Turing Machines). It also allows experimentation with grammars and Pumping Theorem, but in this case only for regular languages. The current version of this tool is only available in Spanish.

- DEUSEXMACHINA is a software package that comprises simulations of seven models of computation covered in a companion textbook [21]. It allows students to draw different kinds of automata (such as Markov algorithms, linear bounded automata, Turing Machines, etc.) using nodes and edge icons.
- JFLAP [13] is a package of graphical tools that can be use to aid in learning the basic concepts of FCT. JFLAP allows not only the design of automata (FA, PDA and Turing Machines) but also input grammars and regular expressions and the conversion between them. Features of JFLAP include several conversions from one representation to another such as non deterministic finite automata to deterministic finite automata (DFA) or DFA to minimum state DFA.

There are also simulator programs for more specific classes of automata. Some interesting examples are described below.

- SIMPRES [1] is a simulator for timed Petri nets. It is intended to represent embedded systems, which extend Petri nets by adding data and real-time information to tokens, and associating functions and delays to transitions.
- TAGS [9] is a software tool designed to define and run Moore and Mealy transducer automata (MTAs). It is a standalone program that allows the user to define and execute MTAs, as well as some conversions from one kind of MTA to another.

We found that many students were quite enthusiastic in trying different simulators for the same kind of automata rather than concentrating on a single one. Learning how to make use of different simulation software for solving the same problem led to questions and problematic situations which incentivated self-directed learning.
Using simulators as teaching aids encouraged also different "views" for simulating a given automaton [19, 7]: a formal view (with an emphasis on the graph-like representation of the automaton), an input-output view (with an emphasis on the computation process), etc. The existence of multiple views for the same automaton helped students to find a proper abstraction level in different problem solving situations. As discussed in [14], a proper handling of such abstraction levels is particularly important in the context of analyzing students' mental processes. In this respect simulators help students to "zoom in and out" when working on practical exercises. It must be stressed that all these simulators were always used as aiding tools without making them a central issue in the FCT course.

[1] See `www.ida.liu.se/~luico/`.

Using Students' Previous Knowledge: Relating FCT to Programming Languages

In most CS curricula, undergraduate students in a FCT course have already taken courses in algebra, discrete mathematics and principles of programming. Most FCT textbooks [18, 21, 15] emphasize the importance of a good mathematical background for a proper understanding of several important FCT notions. Nevertheless, it is quite uncommon to find concrete references or examples linking FCT concepts with actual programming languages (such as PASCAL or JAVA) which are already familiar to students in a FCT course.
In our opinion, keeping FCT unrelated to actual programming languages is dangerous as it leads to a lack of interest and motivation in many CS students, who perceive the treatment of foundational aspects of computer science as too "math-biased". In our FCT course we have brought in many examples which relate the PASCAL programming language (already mastered by the undergraduate students taking the course) to some abstract concepts. Consider for example the definition of recursive language:
Given an alphabet Σ*, a language* L *is recursive in* Σ *if and only if for any string* $w \in \Sigma^*$ *there exists an effective procedure to decide if* $w \in L$ *or* $w \notin L$.
Rather than using an abstract formal language for illustrating this concept for the first time, we suggest our students to consider the formal language **ValidPrograms**=$\{w \mid w$ is a valid PASCAL program$\}$, defined on the alphabet of all valid ASCII characters. In other words, **ValidPrograms** is the (potentially infinite) set of all those strings $\{w_1, w_2, \ldots, w_k, \ldots\}$ such that every w_i is the code of a PASCAL program that can be successfully compiled. Students are then confronted with the question *"Is* **ValidPrograms** *a recursive language?"*. The answer is yes, and the argument is simple: should this not be the case, then the PASCAL compiler wouldn't be able to decide for some particular text file T whether T is a valid PASCAL program or not.
In the same line of reasoning we consider the property that recursive languages are closed under complement. This can be formally proven as in [18]. We found it useful to motivate our students first with a sample recursive language such as PASCAL by posing the following question: *"Is it possible to design an anti-*PASCAL *compiler, i.e. a compiler that rejects*

every valid PASCAL *program, and accepts every ill-formed one?"*. Many students come up with an answer almost immediately: it just suffices to change the message on the screen, replacing the text *"Compilation successful'* for *'Error in program'*, and vice versa. This very idea is the one which underlies the proof of the above property, so the formal proof which is presented later turns out to be quite natural.

The language **ValidPrograms** can be also analyzed in the context of the Chomsky hierarchy. Clearly, **ValidPrograms** is not regular (as sequences of parentheses in PASCAL expressions should be balanced, and detecting such sequences has already been shown to be beyond the power of regular languages). Is **ValidPrograms** context-free? After some guided examples students are induced to think about the fact that every identifier in a PASCAL program has to be declared before it can be used. This characteristic suggests that the set of all strings in **ValidPrograms** cannot be captured with a context-free grammar. After this analysis, the fact that **ValidPrograms** is not context-free is formally shown using properties of closure and homomorphism for context-free languages [18].

Relating FCT to an actual programming language has proved to be very motivating for introducing and relating many abstract concepts of the theory of formal languages. Although our FCT course is not concerned with the design and construction of compilers (a complete course devoted to these topics is to be taken later by the students), some ideas concerning the differences between lexical, syntactic and semantic aspects of programming languages come up naturally in many discussions and dialogues with students.

When discussing the power and limitations associated with the different classes of formal grammars, it is useful to highlight characteristics of PASCAL that can be captured by languages corresponding to each level of the Chomsky hierarchy. For instance, students recognize that Backus-Naur diagrams, commonly used to define the syntax of PASCAL statements, are equivalent to context-free grammars (type 2 in Chomsky hierarchy). They also realize that not every derivation tree describing a syntactically valid program corresponds to a semantically valid program. In theory, semantic aspects of PASCAL could be specified using a context-sensitive grammar (type 1 in Chomsky hierarchy), but such approach would be excessively cumbersome in practice. As a consequence, such aspects are usually specified using natural language statements of the form *"identifiers must be declared before they are used"*, *"actual arguments must agree with formal arguments, both in number and type"*, and so on.

Although of little practical value, it is illuminating for students to see a few representative examples that illustrate the correspondence between the mentioned semantic issues and some abstract languages studied in class. Pushing forward the relation between aspect of PASCAL and different levels of the Chomsky hierarchy, we also bring in the fact that the computing power of PASCAL is equivalent to the one of Turing machines, and hence PASCAL programs are as powerful as Structured Phrase Grammars (type 0 in Chomsky hierarchy). After engaging in discussions in this spirit, the following conclusions become readily perceived by students:

- When analyzing the set of valid tokens for a given programming language, the power of Regular Languages is required. Thus, operators, delimiters, valid numbers, valid identifiers and reserved words in PASCAL can be represented as regular expressions.
- When analyzing a language given by a set of strings corresponding to syntactically valid programs we need the power of Context-Free Languages, as many elements in programming languages work pairwise (e.g. the reserved words BEGIN and END in PASCAL). The well-known first Pumping Lemma [18] can be used to show that some structured statements are non-regular. As an example, the language of nested *"if-then-else"* statements can be shown to be non-regular, but it can be specified using a context-free grammar.
- The power of Context-Sensitive Languages is needed when semantic issues are involved, like for example the concordance between declaration and usage of programming entities. Thus, in a PASCAL program an identifier can only be used if it has been previously declared. Analogously, non-predefined types need to be defined before they are used in a declaration, and functions or procedures cannot be invoked unless declared. The language $L=\{ww \mid w \in \{a,b\}^*\}$ can be seen as an abstraction of the situations mentioned above. This is a prototypical case of context-sensitive but not context-free language.

In dialogues with our students we informally show them the relationship between the above issues and concepts that they will study in more detail later during a Compiler Construction course. Thus, building automata to recognize well-formed tokens is related to writing a *Lexical Analyzer* in a Compiler Construction course. Context-Free grammars are also linked to the role of *Syntactic Analyzer* in a compiler. Finally, we also discuss briefly how to specify context sensitive requirements of a programming language by using attribute-grammars or table-driven models and an associated *Semantic Analyzer*. Students willing to look further into these notions can be referred to LEX/YACC, a set of tools that facilitate the construction of compilers through the use of high-level specifications. These tools combine two components,

a lexical analyzer generator called LEX [17] and a parser generator known as YACC (Yet Another Compiler Compiler)[16]. LEX operation is guided by extended regular expressions while YACC accepts specifications in a restricted form of context free grammar (LALR(1) with disambiguating rules). In addition, YACC facilitates the incorporation of semantic aspects.

It is also helpful to consider the limitations of real programming languages in the light of the limitations of Turing machines. The most fundamental example of such limitations is the unsolvability of the halting problem. However, postulating other examples in terms of programming languages rather than in terms of Turing machines helps to bring into life the topic of noncomputability. For example, students can be presented with problems of the kind *"Is it possible to write a* PASCAL *Program that determines if any two arbitrary* PASCAL *Programs produce exactly the same output given the same input?"* or *"Is it possible to write a* PASCAL *Program that given an arbitrary* PASCAL *Program containing a procedure determines if the procedure will be eventually invoked?"*.

By recognizing that the answer to the above questions is negative, students can appreciate the practical significance of the notion of unsolvability. They discover that unsolvable problems can arise in natural and familiar scenarios, like those of programming languages, and that these problems have remarkable implications, for example the impossibility of constructing a general purpose "automatic PASCAL debugger". The incorporation of such examples helps students become aware of natural connections existing between concrete languages that have a clear practical value (like any programming language) and abstract languages studied in the course.

Reinforcement and Extension Activities: Research articles on FCT

Following Aebli [1], we introduced a number of reinforcement and extension activities at different stages during the FCT course. Different technical articles related to applications of FCT concepts in real-world problems were presented and discussed in class. After reading these articles, students were asked to manifest their opinions and viewpoints in an open dialogue. Some of the articles introduced were the following:

- In [22] the authors analyze the possibility of implementing a hypercomputer, a super-Turing machine capable of going beyond the Turing limit. As pointed out in the article, hypercomputers are still to date purely theoretical devices. Nevertheless, the field of hypercomputation encompasses several open questions for CS, such as whether super-Turing machines can pave the way toward AI, or whether thinking is more than just computing.
- In [6] the author discusses the limits of what technology can do. This article is particularly interesting as it stresses the importance of a good theory and the role of mathematical abstractions in FCT as a foundation for building practical applications and envisioning the future of CS (such as DNA computing and super recursive computation).
- Dasher[2] is an information-efficient text-entry interface [23], driven by natural continuous pointing gestures. Dasher is a competitive text-entry system wherever a full-size keyboard cannot be used. Dasher's behavior can be partly modelled by a finite state automaton for word recognition.
- In [20], the author describes several techniques used in the ongoing war between viruses and anti-viruses. The author refers to the impossibility of creating an invulnerable anti-virus program, emphasizing that even the most sophisticated anti-virus technologies can be misled by certain types of polymorphic virus. The proof of such a claim resembles the proof of the Turing halting problem.

Our experiences showed that students felt very motivated to participate in this kind of discussions. In our opinion, the source for this motivation was twofold: on the one hand, students could perceive many FCT topics as part of the foundational knowledge required to understand many real-world applications. On the other hand, they felt stimulated as their knowledge on FCT allowed them to make a comprehensive reading of technical texts intended for CS professionals.

4. CONCLUSION

Teaching FCT is both a motivating and a challenging task, in which students will be confronted for the first time with fundamental questions such as *"What is computation?"* and *"What kinds of things are computable?"*. Clearly, a good knowledge of the mathematical background underlying the theory of computing is required to be able to understand how to answer such questions. However, as we have discussed in this paper, many CS students do not feel as motivated and interested in mastering these topics as we expected. Their lack of motivation was due not only to the fact that they perceived the treatment of FCT topics as too 'biased' towards mathematics rather than computer science but also to the complexity of the topics themselves. As a result, those students tend to apply some FCT concepts mechanically rather than developing a significant learning of them.

To deal with the above problem, this paper presents a number of didactic strategies that were applied during several semesters in a second-year undergraduate

[2] See `www.inference.phy.cam.ac.uk/dasher/`

FCT course. Those strategies were based on the constructivist model and were expressed in a stronger use of the informatics technology. Our proposal is not to adopt an 'informal' approach to FCT, simplifying the underlying mathematical background. Instead, we tried to make FCT topics more interesting and attractive for our students. It must be pointed out that the different described teaching strategies were simultaneously integrated into different activities. Each activity stressed the application of one of those strategies without excluding the others.

The results have been highly satisfactory. Many of the 'hard' topics of the curricula (such as those concerning the theory of recursive languages, which originally did not seem of much interest for many students during the lectures) turned out to be the source of interesting discussions of ideas. We contend that combining strategies and emphasizing the use of informatics technology is a good complement to the traditional FCT curricula that helps students achieve more significant learning.

5. REFERENCES

[1] H. Aebli. *Denken: Das Ordnen des Tuns - Kognitive Aspekte der Handlungstheorie / Denkprozesse*. Klett-Cotta, 1980.

[2] M. Ben-Ari. Constructivism in computer science education. *Journal of Computers in Mathematics & Sc. Teaching*, 20(1):45–73, 2001.

[3] J. Bruner. *Toward a Theory of Instruction*. Harvard Univ. Press, 1966.

[4] J. Bruner. *Actual Minds, Possible Worlds*. Harvard Univ. Press, 1986.

[5] J. Bruner. *Act of Meaning*. Harvard Press, 1990.

[6] M. Burgin. How We Know What Technology Can Do. *Comm. of the ACM*, 44(11):83–88, 2001.

[7] C. I. Chesñevar, M. L. Cobo, and W. Yurcik. Using theoretical computer simulators for formal languages and automata automata theory. *ACM SIGCSE Bulletin*, 35(2):33–37, 2003.

[8] R. Davis, C. Maher, and N. Noddings. *(Eds.) Constructivist views of the teaching and learning of mathematics*. Nat. Council for Teaching Mathematics, 1990.

[9] A. Esmoris and C. Chesñevar. Una herramienta para la simulación de autómatas traductores en la enseñanza de teoría de la computación. In *Proc. IX Argentinian Congress in Computer Science*, pages 287–295, 2003.

[10] F. Estrebou, M. Lanza, V. Mauco, R. Barbuzza, and L. Fabre. Minerva: Una herramienta para un curso de lenguajes formales y autómatas. In *Proc. of the Lat. Conf. in Informatics*, 2002.

[11] L. D. Fink. *Creating Significant Learning Experiences : An Integrated Approach to Designing College Courses*. Jossey-Bass, 2003.

[12] S. Glynn, R. Yeany, and B. Britton. *(eds.) The Psychology of Learning Science*. L. Erlbaum Ass., 1991.

[13] E. Gramond and S. Rodger. Using jflap to interact with theorems in automata theory. *ACM SIGCSE Bulletin*, 31(1):336–340, 1999.

[14] O. Hazzan. Reducing abstraction level when learning computability theory concepts. In ACM, editor, *Proc. of the ITICSE*, pages 156–160. ACM, 2002.

[15] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Pearson Education, 2001.

[16] S. C. Johnoson. Jacc - yet another compiler compiler. Technical report, 1975. CSTR-32, AT&T Bell Laboratories.

[17] M. E. Lesk. Lex - a lexical analyzer generator. Technical report, 1975. Nr. 39, Bell Laboratories.

[18] H. Lewis and C. Papadimitriou. *Elements of the Theory of Computation (2nd.Ed.)*. Prentice Hall, 1998.

[19] J. McDonald. Interactive pushdown automata animation. *SIGCSE Bulletin*, 34(1):376–380, 2002.

[20] C. Nachenberg. Computer virus-antivirus coevolution. *CACM*, 40(1):46–51, 1997.

[21] G. Taylor. *Models of Computation and Formal Languages*. Oxford Univ. Press, 1998.

[22] C. Teuscher and M. Sipper. Hipercomputation: Hype or Computation. *Communications of the ACM*, 45(8):23–24, 2002.

[23] D. J. Ward and D. J. C. MacKay. Fast hands-free writing by gaze direction. *Nature*, 418(6900):838, 2002.

[24] C. Yehezkel, W. Yurcik, and M. Pearson. Teaching Computer Architecture with a Computer-Aided learning Environment: State-of-the-Art Simulators. In *Proc. Int. Conf. on Simulation and Mult. in Eng. Education*, 2001.