TOWARDS AUTOMATED MANAGEMENT OF COMPILER ASSIGNMENTS

Leena Salmela, Jorma Tarhio and Timo Montonen
Department of Computer Science and Engineering, Helsinki University of Technology, P.O.Box 5400, FI-02015 TKK, Finland

Keywords: Courseware, Computer science, Compiler assignments.

Abstract: We have implemented two software systems for managing compiler assignments in computer science. The first system is a graphical environment on the Web for doing home assignments related to finite state automata and parsers. It also includes an automatic assessment system for the assignments. The second system manages compiler projects. It generates a personalized language for each student and provides a testing tool for the student's compiler. We introduce the main features of the systems and review some experiences.

1 INTRODUCTION

We consider the application of information technologies in teaching a computer science course on compiling of programming languages. Such a course belongs to the computer science curriculum of many universities. Recently we reorganized our compiler course. We used to have one graduate course, which has now been split into two courses. The first one is an introductory course for undergraduate students. It deals mainly with the front-end of a compiler. The second course is for graduate students. It covers advanced topics on the back-end beyond the scope of the original course. The change also created an opportunity to reorganize the assignments of the first course.

We use home assignments in our course. In our former system, a teaching assistant graded the written answers submitted by the students. Several problems arose with this approach. First of all, with our resources it was not possible to give individual feedback to one hundred students, and the delay of the feedback was two weeks, which was too long. Plagiarism has also been a growing concern in the course since all students had the same assignments. Some students also found the previous practice of returning the home assignments as text files in a specified form unintuitive.

To attack these problems we automated the home assignments. Automatic assessment allows immediate feedback to the students and it is possible to give individual assignments to students thus alleviating the problem of plagiarism. Our system is called ACE which is short for Automated Compiler Exercises. ACE works on the Web. The preliminary version of ACE was introduced in the work-in-the-progress paper (Salmela and Tarhio, 2004). Here we deal with the production version, which has been used four times. The main features of ACE are explained in Section 2. In Section 4 we review some experiences of ACE.

Besides home assignments, there is an obligatory project in the course. In our former graduate course, students worked in pairs and implemented a small compiler with register allocation mostly by hand. The assignment was the same for all students. Because the project of an undergraduate course cannot be equally demanding, we dropped register allocation and other advanced features and shifted to extensive use of a compiler writing system. In order to decrease the workload of the teaching assistant, we implemented a tool called TKKCC for managing the projects. TKKCC is short for TKK[1] Compiler Construction. Because TKKCC is able to generate a personalized language for each student, it was natural to shift to individual assignments to alleviate plagiarism. TKKCC also provides a testing tool for a student's compiler. So far TKKCC has been used only once in our course. The details of TKKCC are introduced in Section 3 and experiences are reviewed in Section 4.

[1]The official abbreviation of Helsinki University of Technology is TKK, which comes from the Finnish name Teknillinen korkeakoulu.

Salmela L., Tarhio J. and Montonen T. (2009).
TOWARDS AUTOMATED MANAGEMENT OF COMPILER ASSIGNMENTS.
In *Proceedings of the First International Conference on Computer Supported Education*, pages 243-249
DOI: 10.5220/0001839602430249

2 ACE

2.1 Background

ACE contains a Web-based graphical environment for studying and completing the assignments. The visualizations are mostly adapted from JFLAP (Cavalcante et al., 2004), which is a visualization tool for formal languages and automata theory. JFLAP is based on earlier work of Susan Rodger.

Automatic assessment has been successfully used in introductory courses at our university (Malmi et al., 2003). For example, the Ceilidh system (Benford et al., 1993) and Scheme-Robo (Saikkonen et al., 2001) have been used in the programming courses, the TRAKLA2 system (Korhonen et al., 2003), which has a graphical interface for doing algorithm simulation exercises, has been used in the Data Structures and Algorithms course, and the Stratum framework (Janhunen et al., 2004) has been used in several courses in our university. Automatic assessment has proved to be effective in these cases and the student response has also been generally positive.

There are a number of tools visualizing at least some parts of a compiler (Boroni et al., 2001; Cavalcante et al., 2004; Tscherter et al., 2002; Resler and Deaver, 1998; Khuri and Sugano, 1998; Kerren, 1999; Vegdahl, 2001). Several visualizations of finite automata and parsers have been developed. Some of these visualization tools, like JFLAP (Cavalcante et al., 2004) and Exorciser (Tscherter et al., 2002), have taken a step towards automatic assessment. They allow students to try building their own solutions. When the student is ready, the tool will assess the solution and tell the student if it is right. These tools also allow the student to take a look at the model answer. However, these tools do not fully cover the assignments we have used and they are intended for self study so that they do not keep track of students' points and solutions. Solution building and verifying the solution are fully separated in ACE. This makes it possible to embed ACE in a framework supporting automatic assessment and grading.

2.2 Overview of the Assignments

The assignments of our undergraduate course have been organized into three rounds. In our former graduate course, they were the first three rounds out of six. They deal with finite state automata (FSAs), LL parsing, and LR parsing, respectively. The FSA round has four assignments and the other rounds have five assignments.

In the first assignment of the FSA round the student is given a regular expression and the task is to form a nondeterministic finite state automaton (NFA) using Thompson's construction. Then in the second assignment the constructed NFA is simulated with a given input. In the third assignment the NFA is converted to a deterministic finite state automaton (DFA) and this DFA is then simulated in the last assignment of this round.

The second round deals with LL parsing. First the student should remove left recursion from a given grammar. In the second assignment this grammar is left factored. Then in the third assignment the First and Follow sets needed in the LL parse table construction are calculated. The parse table is filled and in the last assignment the constructed parser is simulated with a given input.

In the third round an LR parser is constructed. In the first assignment the student forms the LR(0) item sets of a given grammar and figures out the transitions between them. Then in the second assignment the First and Follow sets are calculated. Based on these sets also the LR parse table is constructed. The grammar used in this assignment is ambiguous so the parse table now contains ambiguity. In the fourth assignment of this round the ambiguity is removed from the parse table so that given precedence and associativity constraints are satisfied. In the last assignment the constructed parser is simulated with a given input.

Some of these assignments are clearly algorithm simulation exercises. The simulation of an FSA or a parser clearly falls into this category. The solution to this kind of assignment is an ordered list of steps. Some of the other assignments include simulation of an algorithm but the algorithm is more loosely defined. For example, the Thompson's construction algorithm does not define a total order for the construction of the automaton parts. Thus it only defines a partial order for the steps that are needed to construct the whole automaton. Of course a total order may be enforced in such an algorithm but this would unnecessarily complicate the assignment. Some of the assignments are even more loosely defined like the removal of left recursion from a grammar. In this case some transformation rules are presented in the study material but the use of exactly these rules is not enforced. These assignments are conceptual in nature. They test the student's understanding of the concept rather than knowledge of a specific algorithm.

We have ten assignment sets for the first round and nine assignment sets for the last two rounds. Assignments for each student are chosen randomly among those sets. Moreover we allow permuting and replacing of local strings and names in the assignments in

order to artificially increase the number of different assignments. We are also studying ways to generate new grammars and regular expressions for assignments.

2.3 Implementation

Architecture. Given the various types of assignments that the system needs to support, we decided to build a client for the students to do the assignments in a computer-aided manner and verifiers for checking them. These components can then be embedded into a framework which takes care of submissions and the needed book keeping. We call the client with the verifiers ACE.

Overviews of the client and the verifiers are given below. We have embedded the client and the verifiers in the Stratum framework (Janhunen et al., 2004) which follows the client/server architecture. Records on assignments, submissions, and results are kept on the server.

Stratum provides a personal Web page for each student. The ACE client is embedded as an applet to this Web page. This applet is the graphical environment for doing the assignments and the student also submits the solutions with the applet. The personal Web page of the student also shows personal assignments of the student and the submission status of each subtask. The verifiers module is embedded in the server.

ACE Client. The central part of the ACE client is the visualization of the data structures needed in the assignments. Most of these features are present in JFLAP (Cavalcante et al., 2004). Thus the ACE client was built reusing the code from JFLAP. However, some changes also needed to be done. JFLAP does not support showing precedence and associativity information of operators, and so visualization for this was built. The simulation of FSAs and parsers in JFLAP are merely animations, and thus we needed to add some interaction so that the students can show how an FSA or parser works. For example, when simulating an LL parser the student has two choices in each step: to advance in the input or to apply a rule from the parse table.

Another major change was adding the notion of assignment rounds and assignments. Now ACE can lead the student through an assignment round one assignment at a time. Other changes included the design of a new file format which contains the information about assignments and assignment rounds. Because of the new file format it is also not so easy for the students to use JFLAP to generate the correct answers. The generation of correct answers was of course disabled from the user interface.

Fig. 1 shows a screenshot of the ACE client. Here a NFA is being converted to a DFA. The student has already defined the initial state of the DFA and the state which the DFA enters after reading the symbol 'x' in the initial state. The labels of the DFA states show their corresponding NFA state sets.

Verifiers. We were also able to reuse some parts of JFLAP when building the verifiers. Some of the assignments like removing left recursion from a grammar are not supported by JFLAP so we needed to implement new verifiers for these. The simulation of FSAs and parsers in JFLAP are only animations without the possibility of error so we needed to implement new verifiers for these too.

The verifiers have the following general structure. First they check if the input the student used was the one given in the assignment. Then they check if the student's solution is correct and generate feedback to the student. Most of the verifiers check the student's solution by generating a model answer and comparing that to the student's solution. However, in some cases this process does not provide a way to generate good feedback to the student and thus the solution is checked in a different manner.

For example, the first assignment of the FSA round is checked by reversing Thompson's construction and comparing the regular expressions. This way we are better able to tell the student which part of the automaton is correct and which part could not have been generated by Thompson's construction or is generated for a wrong regular expression.

Also the assignments, where left recursion is removed from a grammar or a grammar is left factored, are checked somewhat differently. First the verifier checks that the undesirable elements have been removed from the grammar and then it checks that the new grammar produces the same language as the original one. Note that it is intractable to check if two context free grammars produce the same language but in our case the problem is decidable if we assure that both of the grammars are LL(k) grammars (Rosenkrantz and Stearns, 1970; Olshansky and Pnueli, 1977).

3 TKKCC

Our compiler course has a mandatory project. Each student is given a small language, and the goal of the project is to implement a compiler for it. The student builds a compiler with the Java version of the Coco/R

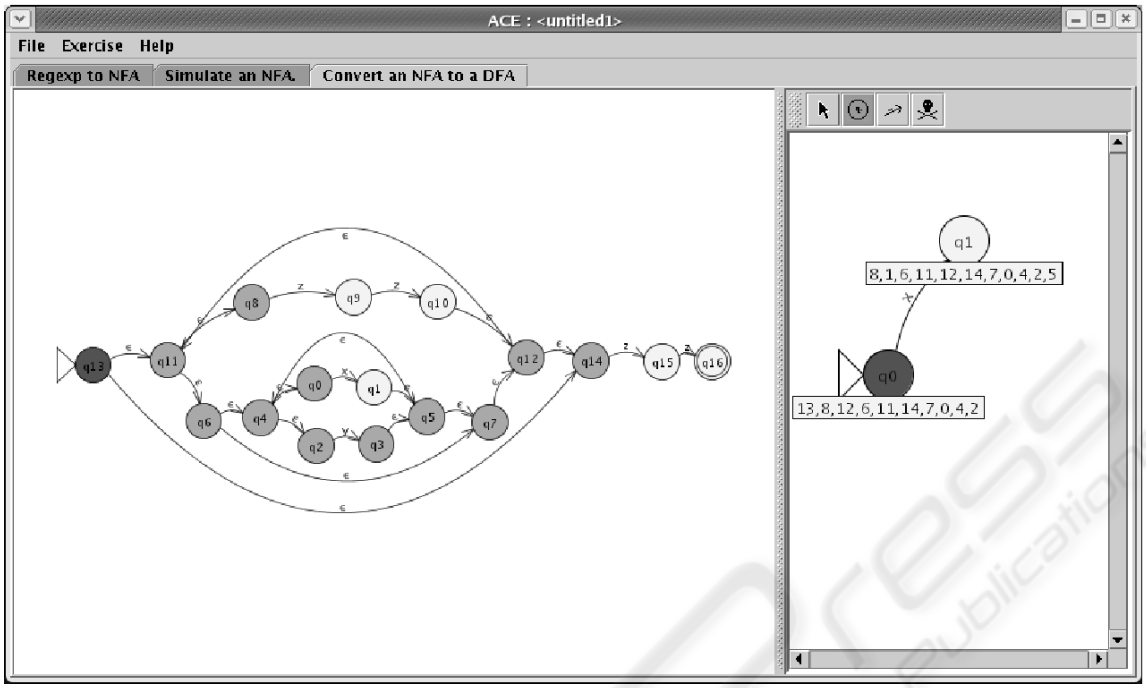

Figure 1: A screenshot from the ACE client. Here the user is converting a NFA to a DFA.

compiler generator (Mössenböck et al., 2008). The compiler should generate code for a virtual stack machine so that the code could be executed with a given interpreter. The student's task is to write an attribute grammar with a few Java classes for the language. The attribute grammar includes code generation.

We implemented a support system called TKKCC for managing projects of our course. TKKCC has three parts:

- **Language generator** creates automatically a personalized language for each student. We designed a base language called Consensus. In generation, a set of its features is selected to the student's language.
- **Test generator** prepares tests for the student's language. We have a large collection of handmade test programs for Consensus, and the test generator translates these programs to the student's language. Expected output is associated with each test program.
- **Test driver** uses the student's compiler to compile test programs and then executes the generated code. The test driver then shows the real output and the expected one. Because the test driver executes code written by students, we have used the Java security manager to restrict access to the system.

As in the case of ACE, we use the Stratum framework (Janhunen et al., 2004) to manage the submissions and keep track of the different assignments of students. The Stratum framework calls the language generator and test generator when the project assignment is created for the students. The students can then access their assignment and related test files and submit the project on the Web. When the student submits a project, Stratum then calls the test drivers to automatically run the tests. The teaching assistant can then use the tools provided by Stratum to check on the progress of the students.

The Consensus language has integer and Boolean data types and integer arrays. Control structures include the if statement and the loop structure. Consensus has functions with parameters. The student's language is a subset of Consensus with syntactic variation. Each element of Consensus has several syntactic alternatives. For example, Consensus has four forms of the loop structure:

- **do** Statement **while** (Expression)
- **repeat** Statement **until** (Expression)
- **while** (Expression) **do** Statement
- **for** (Statement; Expression; Statement) Statement

In a way, the interpreter of the target code is a part of TKKCC, because it has been integrated with the test driver. However, the interpreter naturally depends on the target code used.

The language generator was implemented with templates of the FreeMarker library (FreeMarker, 2008). The language is given to the student as a context-free grammar. The test driver was implemented with Groovy (Groovy, 2008), which is a flexible script language.

Related Systems. There are a few earlier systems that are related to the principles of TKKCC. VCOCO (Resler and Deaver, 1998) is another instructional tool developed for Coco/R but its objective is different from TKKCC. VCOCO visualizes LL(1) parsers generated by the C version of Coco/R. VCOCO shows corresponding parts in the grammar, generated code, and input. Unfortunately, VCOCO does not work with the Java version of Coco/R and thus could not be used in our course. Otherwise it would have been useful for our students.

ART (Aycock, 2003) is a tool which helps to create more complex test programs for compilers written as course projects. In this kind of projects the compiled language is often a simple toy language which lacks more complex structures like arrays. This makes it difficult to construct complex programs suitable for stress testing students' compilers. ART generates such complex input programs by translating arrays of fixed size to single variables and replacing array references by references to the variable corresponding to that item. ART could be used with TKKCC to create more complex test programs. However, because the Consensus language of TKKCC has arrays, it is not so tedious to create complex test programs.

Cool (Aiken, 1996) is a support system for compiler projects. Cool provides a portable compiler project, a class library, which contains common routines of a compiler, a reference compiler and a language reference manual for the input language of the compiler project. Cool has modular assignments that have been designed to be independent and thus a student is able to continue the project even if an earlier phase has failed.

4 EXPERIENCES

4.1 ACE

The main advantage of ACE is the significant decrease in the workload of the teaching assistant, because ACE delivers and assesses the assignments as well as gives feedback to the students. The time needed for yearly maintaining is short.

There are three assignment rounds (involved with the front-end of the compiler) in the course. In order to pass the course students have to pass two of these rounds. Doing more than the required number of assignments is voluntary and does not give any extra credit to the students. After adopting ACE the percentage of students who completed all three rounds increased from 37% to 51% (the figures are averages over several years).

When ACE was used for the first time, we asked the students of the course to answer a questionnaire, and 60% of the 91 students replied. Concerning the possible advantages of ACE, the most common answer dealt with immediate feedback, which helped students to correct their mistakes. Also the ability to make a round incrementally in several sessions and the possibility to iterate answers were important benefits.

Several students said that ACE saves time when compared with the traditional pen and paper way. However, the students who already mastered the topic of an assignment before using ACE found the computer-aided approach slower than pen and paper, because learning ACE was in a way additional to them. Related to this, we asked whether the students felt wasting time with ACE. Only 13% of them answered yes, and 2% were uncertain. The rest 85% did not waste time. The average use time per round was 4 h 8 min.

We also asked the students if they encountered any problems when using ACE, and 76% of them reported some problems, mostly minor technical problems. All the problems have already been corrected. A typical complaint dealt with vague error messages and unclear and limited instructions. Several students wished for demonstrations and examples in order to get easier acquainted with the system.

Then we asked whether ACE supported learning. The answer was clear: 96% of the students answered "yes".

We requested the students to compare ACE with similar computer-aided tools for homework assignments of other courses. There are at least three other courses using such technology at our department. So 80% of the students had used similar systems earlier. ACE was well received, and almost half of students considered it to be at least better than the other systems. One student explained his opinion as follows pointing out some problems of computer-aided instruction: "ACE is better because it is not as mechanical as some other tools, which do not require pondering at all; yet even ACE provides so much help (in guiding towards the solution) that if only using ACE, one does not know what to do without it. So ACE is

good, but pen-and-paper is still needed."

Among the same student group we made another questionnaire in connection with the course evaluation in which 54% of the students participated. A course evaluation is organized in the end of every course at our department. Besides fixed questions, the teacher is allowed to present additional questions. We asked the students to compare ACE with the traditional approach. The group studied the original version of our course so that they had also three other assignment rounds concerning the back-end of a compiler. These rounds consisted of traditional pen-and-paper assignments. The first questionnaire took place in the middle of the course, whereas the students had completed all the assignments while the second questionnaire was made.

The first question dealt with meaningfulness. Only few of the students had experienced the traditional way more meaningful than ACE. One third did not see any difference between the alternatives, and more than half found the computer-aided way more meaningful.

Next the students were asked to compare the laboriousness of the approaches. Based on the first questionnaire we already knew that those students, who already mastered the subject, wasted time whereas those students, who did not know the subject beforehand, saved time. Now in the end of the course, half of the students found the traditional way more laborious. About one third experienced both approaches similar, and the rest of the students considered ACE more laborious.

The last question was about learning. Half of the students thought that the computer-aided approach was better from the point of view of learning. Only 12% of them found the traditional approach better. The summary of the second questionnaire is given in Table 1.

Table 1: Answers to the question: Which way was more meaningful, more laborious, and better from the point of view of learning? ACE or the traditional approach? The numbers are percentages.

Which way was...	ACE	No diff.	Trad.
More meaningful?	59	33	8
More laborious?	18	31	51
Better from the point of view of learning?	51	37	12

4.2 TKKCC

TKKCC saves the teaching assistant's time but not at all as much as in the case of ACE. TKKCC delivers the assignments automatically. Many students need consulting, and TKKCC cannot help much. It only provides semi-automatic assessment as it only runs tests automatically but the teaching assistant will still have to check other issues like documentation.

From the student's point of view, TKKCC does not offer much additional when compared with conventional use of Coco/R. However, the testing capability is a new service and the students also appreciated this feature.

5 CONCLUDING REMARKS

We have described two systems managing compiler assignments. The ACE system automatically assesses assignments related to finite state automata and parsers. The system supports individual assignments for students and it has a visual interface for studying and completing the assignments. The system has been used four times in our compiler course, and it was well received by the students.

ACE is an advanced learning environment. It makes the concepts of compiler construction concrete by visualizing them. ACE supports learning by doing. The student can submit a trial solution, and the system gives feedback about the possible errors. Because only correct submissions are accepted, the student is allowed to try each assignment of a round several times. Most of the assignments are constructed in such a way that it is almost impossible to reach an acceptable solution only by guessing.

ACE shares a part of the visualizations of JFLAP which is an excellent tool for demonstration and self-study purposes. However, the integrated assessment and bookkeeping make ACE more useful than JFLAP because the assignments of a large course can be delivered and graded automatically with ACE.

ACE works on the Web so that students can do the assignments anywhere at the time suitable for them. Thus ACE suits well to distance learning e.g. in a virtual university. The general principles of ACE can be adapted to other subjects with constructive assignments.

TKKCC manages compiler projects. It generates languages for students, and provides a testing framework for the implemented compiler. We plan to add a new module, which would help students in writing attribute grammars for Coco/R, to TKKCC . This module would recognize some typical errors students make and provide instructions on how to proceed.

Recently we have updated the language generator of TKKCC so that it produces two alternative assignments for each student: easy and full. Arrays, and functions and function calls have been removed from

the easy assignment and the associativity and precedence of operators in expressions have been solved for the student in a trivial way by requiring explicit use of parenthesis in the input programs of the student's compiler. This will allow the weaker students to start working with an easy assignment while still giving them the possibility to switch to the full assignment later. Of course, the highest points cannot be scored with an easy assignment. This feature will be used in our course this fall for the first time.

Both of the tools, ACE and TKKCC, are freely available.

ACKNOWLEDGEMENTS

We thank Susan Rodger and Tomi Janhunen for letting us use their codes. The help of Venla Hytönen is appreciated.

REFERENCES

Aiken, A. (1996). Cool: a portable project for teaching compiler construction. *SIGPLAN Not.* 31(7):19–24.

Aycock, J. (2003). The art of compiler construction projects. *SIGPLAN Not.* 38(12):28–32.

Benford, S., Burke, E., Foxley, E., Gutteridge, N., and Zin, A. M. (1993). Ceilidh: A course administration and marking system. In *Proceedings of the 1st International Conference of Computer Based Learning*.

Boroni, C., Goosey, F., Grinder, M., and Ross R. (2001). Engaging students with active learning resources: Hypertextbooks for the web. In *Proceedings of the 32nd SIGCSE Technical Symposium on Computer Science Education*, pages 65–69. ACM.

Cavalcante, R., Finley, T., and Rodger, S. H. (2004). A visual and interactive automata theory course with JFLAP 4.0. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*, pages 140–144. ACM.

FreeMarker (2008). http://www.freemarker.org/. Oct. 2008.

Groovy (2008). http://groovy.codehaus.org. Oct. 2008.

Janhunen, T., Jussila, T., Järvisalo, M., and Oikarinen, E. (2004). Teaching Smullyan's analytic tableaux in a scalable learning environment. In *Proceedings of the 4th Finnish/Baltic Sea Conference on Computer Science Education*.

Kerren, A. (1999). Animation of the semantical analysis. In *Proceedings of 8. GI-Fachtagung Informatik und Schule, INFOS99*. Informatik aktuell, Springer. (in German).

Khuri, S. and Sugano, Y. (1998). Animating parsing algorithms. In *Proceedings of the 29th SIGCSE Technical Symposium*, pages 232–236. ACM.

Korhonen, A., Malmi, L., and Silvasti, P. (2003). TRAKLA2: a framework for automatically assessed visual algorithm simulation exercises. In *Proceedings of the 3rd Finnish/Baltic Sea Conference on Computer Science Education*, pages 48–56.

Malmi, L., Korhonen, A., and Saikkonen. R. (2002). Experiences in automatic assessment on mass courses and issues for designing virtual courses. In *Proceedings of the 7th Annual SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education*, pages 55–59. ACM.

Mössenböck, H., Löberbauer, M., and Wöß, A. (2008). The Compiler Generator Coco/R. http://www.ssw.uni-linz.ac.at/coco/. Oct. 2008.

Olshansky, T. and Pnueli, A. (1977). A direct algorithm for checking equivalence of LL(k) grammars. *Theoretical Computer Science* 4:321–349.

Resler R. D. and Deaver, D. M. (1998). VCOCO: A visualisation tool for teaching compilers. *ACM SIGCSE Bulletin*, 30(3):199–202.

Rosenkrantz, D. J. and Stearns, R. E. (1970). Properties of deterministic top-down grammars. *Information and Control*, 17(3):226–256.

Saikkonen, R. , Malmi, L. , and Korhonen, A. (2001). Fully automatic assessment of programming exercises. In *Proceedings of the 6th Annual SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education*, pages 133–136. ACM.

Salmela, L. and Tarhio, J. (2004). ACE: Automated compiler exercises. In *Proceedings of the 4th Finnish/Baltic Sea Conference on Computer Science Education*.

Tscherter, V., Lamprecht, R., and Nievergelt J. (2002). Exorciser: Automatic generation and interactive grading of exercises in the theory of computation. In *4th International Conference on New Educational Environments*, pages 47–50.

Vegdahl, S. R. (2001). Using visualization tools to teach compiler design. *Journal of Computing Sciences in Colleges*, 16(2):72–83.