Towards Deeper Understanding of Syntactic Concepts in Programming

Sebastian Gross, Sven Strickroth, Niels Pinkwart, and Nguyen-Thinh Le

Clausthal University of Technology, Department of Informatics
Julius-Albert-Str. 4, 38678 Clausthal-Zellerfeld, Germany
{sebastian.gross,sven.strickroth}@tu-clausthal.de
{niels.pinkwart,nguyen-thinh.le}@tu-clausthal.de

Abstract. Syntactic mistakes and misconceptions in programming can have a negative impact on students' learning gains, and thus require particular attention in order to help students learn programming. In this paper, we propose embedding a discourse on syntactic issues and student's misconceptions into a dialogue between a student and an intelligent tutor. Based on compiler (error) messages, the approach aims to determine the cause for the error a student made (carelessness, misconception, or lack of knowledge) by requesting explanations for the violated syntactic construct. Depending on that cause, the proposed system adapts dialogue behaviours to student's needs by asking her to reflect on her knowledge in a self-explanation process, providing error-specific explanations, and enabling her to fix the error herself. This approach is designed to encourage students to develop a deeper understanding of syntactic concepts in programming.

Keywords: intelligent tutoring systems, programming, dialogue-based tutoring

1 Introduction

Programming is a useful skill and is related to several fields of study as economy, science, or information technology. Thus, teaching basics of programming is part of many curricula in universities and higher education. Programming is often taught bottom-up: First, syntactic aspects and low-level concepts are presented to students (e. g. variable declarations, IF, WHILE constructs, ... in the object-oriented programming paradigm). Then, iteratively higher-level concepts are taught (e. g. methods, recursion, usage of libraries, ...). Learning a programming language, however, cannot be approached theoretically only. It requires a lot of practice for correct understanding of abstract concepts (technical expertise) as well as logical and algorithmic thinking in order to map real-world problems to program code. Studies [8, 17] and our own teaching experiences have shown that studying programming is not an easy task and many students already experience (serious) difficulties with the basics: writing syntactically correct programs which can be processed by a compiler.

Source code is the basis for all programs, since without it algorithms cannot be executed and tested. Here, testing does not only mean testing done by

students themselves. Often tutorial and/or submission systems [7, 18] are used by lecture advisors in order to optimize their workflow and to provide students some further testing opportunities. These tests often focus on the algorithms, check program outputs given a specific input and require runnable source code.

Creating correct source code requires good knowledge and strict observance of the syntax and basic constructs of the programming language. Yet, students often use an integrated development environment (IDE) from the very beginning. Here, code templates and also possible solutions for syntactic errors are offered. Based on our experience over several years of teaching a course on "Foundations of programming" in which Java is introduced and used as a main programming language, we suppose that these features (code templates provided by an IDE) possibly hinder learning and deeper understanding: Novice programmers seem to use these features and suggestions (which are actually addressed to people who already internalized the main syntactic and semantic concepts of programming) blindly. As a result, students are often not able to write programs on their own (e. g. on paper) and do not understand the cause of errors.

In this paper, we propose a new tutoring approach which initiates a dialogue-based discourse between a student and an intelligent tutor in case of a syntactic error. The intelligent tutor aims at detecting a possible lack of knowledge or an existing misconception as well as suggesting further readings and correcting the misconception, respectively. The remainder of this paper is organized as follows: First, in Section 2, we give an overview of the state of the art of intelligent learning systems in programming. In Section 3, we then describe our approach in more detail, illustrate an exemplary discourse, and characterize possible approaches for an implementation. Finally, we discuss our approach in Section 4, draw a conclusion and point out future work in Section 5.

2 Intelligent Learning Systems in Programming

In recent years, Intelligent Tutoring Systems (ITSs) have found their way increasingly into classrooms, university courses, military training and professional education, and have been successfully applied to help humans learn in various domains such as algebra [10], intercultural competence [16], or astronaut training [1]. Constraint-based and cognitive tutor systems are the most established concepts to build ITSs, and have shown to have a positive impact on learning [14]. In the domain of programming, several approaches have been successfully applied to intelligently support teaching of programming skills using artificial intelligence (AI) techniques. In previous work [12], we reviewed AI-supported tutoring approaches for programming: example-based, simulation-based, collaboration-based, dialogue-based, program analysis-based, and feedback-based approaches.

Several approaches for building ITSs in the domain of programming are based on information provided by compilers. The Expresso tool [6] supports students in identifying and correcting Java programming errors by interpreting Java compiler error messages and providing feedback to students based on these messages. JECA is a Java error correcting algorithm which can be used in Intelligent Tutoring Systems in order to help students find and correct mistakes [19]. The

corresponding system prompted learner whether or not the system shall automatically correct found errors. Coull and colleagues [3] suggested error solutions to learners based on compiler messages by parsing these messages and comparing them to a database. These approaches aim to support learners in finding and correcting syntactic errors without explicitly explaining these issues, and, thus, did not ensure that a learner internalizes the underlying concept. HelpMeOut [5], however, is a recommender system based on compiler messages and runtime exceptions which formulated queries to a database containing error-specific information in order to recommend explanations for students' mistakes. The underlying database could be extended by users' input generated via peer interactions. This approach did not allow a discourse in order to determine student's knowledge or to correct possible misconceptions in student's application of knowledge, but provides solutions to students without encouraging students' learning. In our approach, we propose a dialogue-based discourse between a student and a tutor which aims at identifying the cause of the syntactic error, and at ensuring that the student gains a deeper understanding of the underlying syntactic concept she violated.

3 Solution Proposal

Programmers need to master syntactic and semantic rules of a programming language. Using integrated development environments such as Eclipse or Netbeans supports experienced programmers in finding and correcting careless mistakes and typos, and thus help them to efficiently focus on semantic issues. Novice programmers, however, who are still learning a programming language and, thus, are probably not entirely familiar with the syntactic concepts might be overwhelmed by messages provided by compilers. Interpreting error messages and correcting mistakes based on these messages can be a frustrating part of programming for those learners. IDEs, indeed, help them finding and correcting an error, but also impede learner's learning if learners follow IDEs' suggestion without reflecting on these hints and understanding why an error occurred.

How well programmers are able to find and correct syntactic mistakes strongly depends on the quality of messages and hints provided by compilers or IDEs [2, 13, 15]. Following previous work in the field of intelligent supporting systems for programming, we propose to provide guidance to novice programmers based on compiler (error) messages in order to help them master syntactic issues of programming languages. Instead of enriching compiler messages, we aim to determine student's knowledge about a specific violated syntactic construct. Depending on a student's level of knowledge, we propose to adapt the system's learning support to student's individual needs. For this, we distinguish three causes for syntactic errors:

E1 Errors caused by carelessness,
E2 Errors caused by lack of knowledge,
E3 Errors caused by misconceptions.

In order to determine which one of the three causes applies to a specific error, we propose to initiate a discourse between the learner and an intelligent

tutor (shown in Figure 1). Information provided by a compiler can be used to identify an erroneous part and the syntactic concept the student violated in order to lead the discourse to corresponding syntactic aspects. Embedded in dialogues and backed up by a knowledge database, the tutor first aims to determine whether or not the student is able to explain the underlying concept of the violated statement or syntactic expression. Our approach requires a knowledge base of the most typical errors of students. For this purpose, we used data collected in the submission system GATE [18]. We used GATE in our introductory Java teaching courses since 2009. This system supports the whole workflow from task creation, file submission, (limited) automated feedback for students to grading. We analyzed and categorized 435 compiler outputs of failed Java code compilations of student solutions: The ten most common syntax errors according to the compiler outputs (covering 70 % of all errors) are missing or superfluous braces (56 cases), usage of missing classes (e. g. based on an incomplete upload; 45), mismatching class names (according to the file name; 37), usage of undeclared variables (35), problems with if-constructs (23), usage of incompatible types (21), method definitions within other methods (primarily within the main method; 19), usage of undeclared methods (18), missing return statements in methods (14), and problems with SWITCH statements (12).

Just as experienced programmers also novice programmers make mistakes which are caused by carelessness (**E1**, e. g. a typo). In this case students are able to correctly and completely explain the concepts. The tutor then confirms the student's correct explanation, and students are able to fix the error without any further help. Errors caused by lacks of knowledge or misconception in the application of the knowledge, however, require special attention. This is the case if the student is not able to correctly and/or completely explain the underlying concept of a statement or syntactic expression which was violated. Then the tutor is not able to recognize student's explanation and distinguishes whether

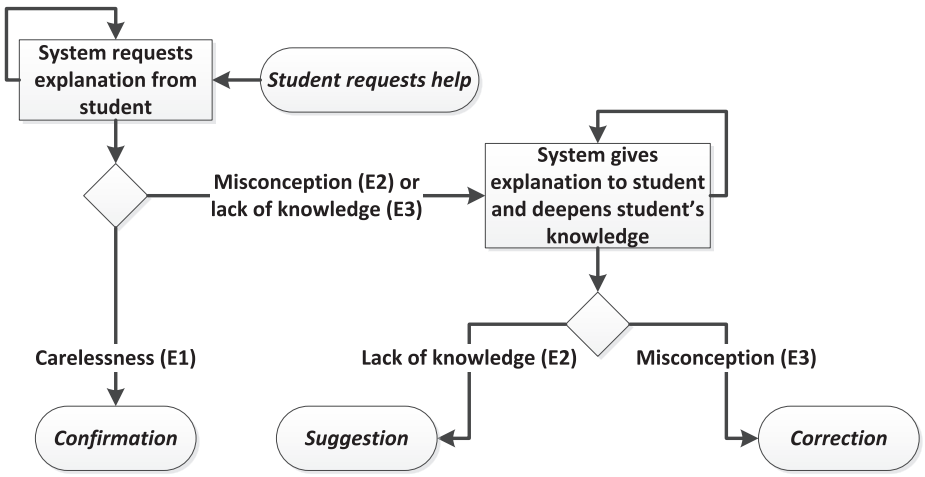


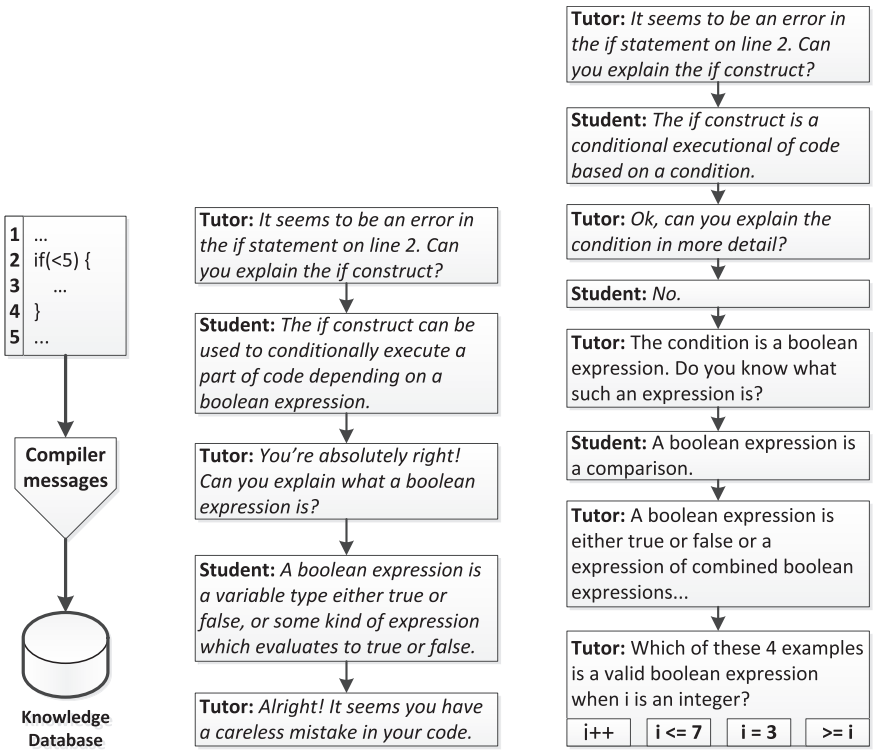
Fig. 1: Dialogue-based identification of cause for syntactic error

a lack of knowledge or a misconception caused the error by requesting further explanations from students. If a lack of knowledge is detected (**E2**), the tutor then suggests how to correct the error or points to the part of a (video) lecture explaining the violated concept. In the other case, if a misconception is detected (**E3**), the tutor changes its role in the discourse in order to revise the student's wrong and/or incomplete explanation. In this error-specific dialogue, the tutor then tries to explain the underlying concept the student violated. Therefore, the tutor could then provide step-by-step explanations using the knowledge base. To evaluate student understanding of single steps of explanations, the tutor could ask the student to confirm whether or not she understood the explanation, to ask her to complete/correct incomplete/erroneous examples covering the underlying syntactic concept, or to assess student's knowledge in question and answer manner.

In summary, we propose a dialogue-based intelligent tutor which initially interprets compiler (error) messages in order to identify the syntactic concept the student violated. Based on the compiler information, the tutor initiates a discourse with the student where it determines the cause of the error (**E1**, **E2** or **E3**). In a deeper examination of student's knowledge, the tutor uses a knowledge base in order to impart and deepen the concept which the syntactic error corresponds to. The tutor uses a computational model that is capable of automatically evaluating student's responses on tutor's questions. The goal is to correct misconceptions or to suggest further readings in order to fill lacks of knowledge and enable students to fix their mistakes in their own this way. In Section 3.2, we explain how such a model can be implemented.

3.1 Exemplary Dialogue-Based Discourse

In the above, we introduced typical syntactic errors that were made by students who attended a course on "Foundations of programming". The dataset contained students' exercise submissions of one of our introductory Java courses. To illustrate our approach (described in Section 3), we discuss a dialogue-based discourse exemplary for one of those typical errors (see Figure 2). A typical error that often occurred in students' submissions was that the implementation of a condition statement (IF construct) did not match the underlying syntactic concept. In the first dialogue (shown in Figure 2b), the tutor asks the student to explain the IF construct and, because it is part of an if-statement, what a boolean expression is. Here, the student is able to explain both concepts, and thus the mistake seems to have been caused by carelessness and the tutor confirms the student's explanations. In the second dialogue (shown in Figure 2c), the student gives an incomplete explanation on tutor's request. The tutor, consequently, asks the student to explain the condition in more detail which the student is not able to do. At that point, the tutor switches from requesting to providing explanations, and aims at deepening student's knowledge. Finally, the tutor aims at evaluating whether the student understood its explanations by asking a multiple-choice-question. Depending on the student's answer, the tutor can then assess whether the error was caused by a misconception or lack of knowledge. In the one case, the student is able to correctly respond to tutor's



(b) Dialogue 1 (c) Dialogue 2

Fig. 2: Dialogue-based discourse between student and intelligent tutor

question which indicates a misconception that could be corrected during the discourse. In the other case, the student is not able to correctly respond to the tutor's question which indicates lack of knowledge. Here, the tutor might suggest the student to repeat appropriate lecture(s)/exercise(s) in order to acquire the necessary knowledge.

3.2 Technical Implementation

In the dialogue-based approach proposed in this paper we need to distinguish two types of student's answers. The first one consists of explanations about a concept upon request of the system, and the second one includes short answers on error-specific examples and questions.

In order to understand a student's explanation on a programming concept we either provide her options to be chosen or allow her to express the explanation in a free form. In the first case, the system can understand the student's explanation by associating each template with a classifier of the error type. For example, in order to determine whether the student has made an error in the IF condition statement by carelessness, by misconception or lack of knowledge,

we can ask the student to explain this concept and provide her with three possible answers: 1) *The IF construct can be used to conditionally execute a part of code depending on a boolean expression.*, 2) *The IF construct can be used to express factual implications, or hypothetical situations and their consequences.*, 3) *I have no idea.* Obviously, the first answer is correct and the second answer is a misconception because students might refer the IF construct of a programming language (e. g., Java) to the IF used in conditional sentences in the English language. The third option indicates that the student has lack of the condition concept. This approach seems to be easy to implement, but requires a list of typical misconceptions of students. If we allow the student to express an explanation in a free form, the challenge is to understand possible multi-sentential explanations. In order to deal with this problem Jordan and colleagues [9] suggested to process explanations through two steps: 1) single sentence analysis, which outputs a first-order predicate logic representation, and 2) then assessing the correctness and completeness of these representations with respect to nodes in correct and buggy chains of reasoning.

In order to understand short answers on error-specific examples and questions, we can apply the form-filling approach for initiating dialogues. That is, for each question/example, correct answers can be anticipated and authored in the dialogue system. This approach is commonly used in several tutoring systems, e. g., the dialogue-based EER-Tutor [20], PROPL [11], AUTOTUTOR [4]. In addition to the form-filling approach, the Latent Semantic Analysis technique can also be deployed to check the correctness in the natural language student's answer by determining which concepts are present in a student's utterance (e. g., AUTOTUTOR).

4 Discussion

Our approach relies on the compiler's output. So, ambiguity of compiler messages is a crucial issue (also for students). The standard Java compiler works by following a greedy policy which causes that errors are reported for the first position in the source code where the compiler recognized a mismatch despite the fact that the cause of the error might lie somewhere else. There are also different parsers that use other policies and are capable of providing more specific feedback (e. g. the parser of the Eclipse IDE). Taking the code fragment "int i : 5;", e.g., the standard Java compiler outputs that it expects a ";" instead of the colon. The Eclipse compiler, however, outputs, that the colon is wrong and suggests that the programmer might have wanted to use the equal character "=". This difference in the compilers becomes even more manifest for lines where an opening brace is included. If there is an error in this line before the brace, the whole line is ignored by the standard Java compiler and a superfluous closing brace is reported at the end of the source code. Here, using a better parser (or even a custom parser) could improve error recognition regarding the position of the error and the syntactic principles violated by the programmer. Additional and more detailed information can help to cover more syntactic issues and to apply a more sophisticated discourse between learners and a dialogue-based tutor.

Generally, it is sufficient for our approach that a compiler reports the correct line and the affected basic structure of an error (e. g. If-statement), since our approach does not aim for directly solving the error, but supporting the students to fix the mistake on their own. This, however, requires a good knowledge base of the basic structures about a programming language.

5 Conclusion and Future Work

In this paper, we proposed a dialogue-based approach interpreting compiler (error) messages in order to determine syntactic errors students made, and thus to adapt the behaviour of the intelligent tutor to the individual needs of students depending on three causes of errors (carelessness, lack of knowledge, or misconception). Our proposed system initiates a dialogue asking for explanations of the violated syntactic construct and determines which cause applies for the affected violated construct. Then the proposed approach adapts dialogue behaviours to student's needs confirming correct knowledge or providing error-specific explanations. We argued that this method works better than just presenting error messages or suggestions for fixing an error, because it encourages students to reflect on their knowledge in a self-explanation process and finally enables them to fix the errors themselves.

In future, we plan to implement our approach and test it with students in an introductory programming course. Initially, we will apply self-explanation in human-tutored exercises in order to gather dialogues which can be used to build a model for our approach.

References

[1] K. Belghith, R. Nkambou, F. Kabanza, and L. Hartman. An intelligent simulator for telerobotics training. *IEEE Transactions on Learning Technologies*, 5(1):11–19, 2012.

[2] B. Boulay and I. Matthew. Fatal error in pass zero: How not to confuse novices. In G. Veer, M. Tauber, T. Green, and P. Gorny, editors, *Readings on Cognitive Ergonomics Mind and Computers*, volume 178 of *Lecture Notes in Computer Science*, pages 132–141. Springer Berlin Heidelberg, 1984.

[3] N. Coull, I. Duncan, J. Archibald, and G. Lund. Helping Novice Programmers Interpret Compiler Error Messages. In *Proceedings of the 4th Annual LTSN-ICS Conference*, pages 26–28. National University of Ireland, Galway, Aug. 2003.

[4] A. Graesser, N. K. Person, and D. Harter. Teaching Tactics and Dialog in AutoTutor. *International Journal of Artificial Intelligence in Education*, 12:257–279, 2001.

[5] B. Hartmann, D. MacDougall, J. Brandt, and S. R. Klemmer. What would other programmers do: suggesting solutions to error messages. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages 1019–1028, New York, NY, USA, 2010. ACM.

[6] M. Hristova, A. Misra, M. Rutter, and R. Mercuri. Identifying and correcting java programming errors for introductory computer science students. In *Proceedings of the 34th SIGCSE technical symposium on Computer science education*, SIGCSE '03, pages 153–156, New York, NY, USA, 2003. ACM.

[7] P. Ihantola, T. Ahoniemi, V. Karavirta, and O. Seppälä. Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, Koli Calling '10, pages 86–93, New York, NY, USA, 2010. ACM.

[8] T. Jenkins. A participative approach to teaching programming. In *Proceedings of the 6th annual conference on the teaching of computing and the 3rd annual conference on Integrating technology into computer science education: Changing the delivery of computer science education*, ITiCSE '98, pages 125–129, New York, NY, USA, 1998. ACM.

[9] P. W. Jordan, M. Makatchev, U. Pappuswamy, K. VanLehn, and P. L. Albacete. A natural language tutorial dialogue system for physics. In G. Sutcliffe and R. Goebel, editors, *FLAIRS Conference*, pages 521–526. AAAI Press, 2006.

[10] K. R. Koedinger, J. R. Anderson, W. H. Hadley, and M. A. Mark. Intelligent tutoring goes to school in the big city. *International Journal of AI in Education*, 8:30–43, 1997.

[11] H. C. Lane and K. VanLehn. A dialogue-based tutoring system for beginning programming. In V. Barr and Z. Markov, editors, *FLAIRS Conference*, pages 449–454. AAAI Press, 2004.

[12] N. T. Le, S. Strickroth, S. Gross, and N. Pinkwart. A review of AI-supported tutoring approaches for learning programming. In *Accepted for the International Conference on Computer Science, Applied Mathematics and Applications 2013, Warsaw, Poland*. Springer Verlag, 2013.

[13] G. Marceau, K. Fisler, and S. Krishnamurthi. Measuring the effectiveness of error messages designed for novice programmers. In *Proceedings of the 42nd ACM technical symposium on Computer science education*, SIGCSE '11, pages 499–504, New York, NY, USA, 2011. ACM.

[14] A. Mitrovic, K. Koedinger, and B. Martin. A comparative analysis of cognitive tutoring and constraint-based modeling. In P. Brusilovsky, A. Corbett, and F. Rosis, editors, *User Modeling 2003*, volume 2702 of *Lecture Notes in Computer Science*, pages 313–322. Springer Berlin Heidelberg, 2003.

[15] M.-H. Nienaltowski, M. Pedroni, and B. Meyer. Compiler error messages: what can help novices? In *Proceedings of the 39th SIGCSE technical symposium on Computer science education*, SIGCSE '08, pages 168–172, New York, NY, USA, 2008. ACM.

[16] A. Ogan, V. Aleven, and C. Jones. Advancing development of intercultural competence through supporting predictions in narrative video. *International Journal of AI in Education*, 19(3):267–288, Aug. 2009.

[17] A. Robins, J. Rountree, and N. Rountree. Learning and teaching programming: A review and discussion. *Computer Science Education*, 13(2):137–172, 2003.

[18] S. Strickroth, H. Olivier, and N. Pinkwart. Das GATE-System: Qualitätssteigerung durch Selbsttests für Studenten bei der Onlineabgabe von Übungsaufgaben? In *DeLFI 2011: Die 9. e-Learning Fachtagung Informatik*, number P-188 in GI Lecture Notes in Informatics, pages 115 – 126. GI, 2011.

[19] E. R. Sykes and F. Franek. Presenting jeca: a java error correcting algorithm for the java intelligent tutoring system. In *Proceedings of the IASTED Conference on Advances in Computer science*, 2004.

[20] A. Weerasinghe, A. Mitrovic, and B. Martin. Towards individualized dialogue support for ill-defined domains. *International Journal of AI in Education*, 19(4):357–379, Dec. 2009.