Informatics in Education, 2007, Vol. 6, No. 1, 139–152

Novices' Progress in Introductory Programming Courses

Linda MANNILA

Dept. of Information Technologies, Åbo Akademi University, Turku Centre for Computer Science
Joukahaisenkatu 3-5 A, 20520 Turku, Finland
e-mail: linda.mannila@abo.fi

Received: August 2006

Abstract. This paper presents an approach for educators to evaluate student progress throughout a course, and not merely based on a final exam. We introduce *progress reports* and describe how these can be used as a tool to evaluate student learning and understanding during programming courses. Complemented with data from surveys and the exam, the progress reports can be used to build an overall picture of individual student progress in a course, and to answer questions related to how students (1) understand program code as a whole, (2) understand individual constructs, and (3) perceive the difficulty level of different programming topics. We also present results from using this approach in introductory programming courses at secondary level. Our initial experience from using the progress reports is positive, as they provide valuable information during the course, which most likely would remain uncovered otherwise.

Key words: introductory programming, learning programming, program understanding, SOLO study.

1. Introduction

In (Grandell *et al.*, 2006), we reported on a study from teaching introductory programming at high school level.[1] The results showed that abstract topics such as algorithms, subroutines, exception handling and documentation were considered most difficult, whereas variables and control structures were found rather straight forward. These results were in line with those of other researchers (e.g., (Haataja *et al.*, 2006; Lahtinen *et al.*, 2005)). In addition, our findings indicated that most novices found it difficult to point out their weaknesses. Moreover, exam questions asking the students to read and trace code showed a serious lack in program comprehension skills among the students. One year later (2005/2006) we conducted a new study, in which we further investigated these issues using what we have called *progress reports*. This paper presents the results from this study.

[1] In the Finnish educational system, high schools are referred to as upper secondary schools, providing education to students aged 16–19. The main objective of these schools is to offer general education preparing the students for the matriculation examination, which is a pre-requisite for enrolling for university studies.

We begin the paper with a background section, followed by a section describing the study and the methods used. Next, we present and discuss the results, after which we conclude the paper with some final words and suggestions for how the ideas presented can be implemented in practice.

2. Background

Introductory programming courses tend to have a strong focus on construction, with the overall goal to get students to write programs as quickly as possible using some high-level language. This is understandable as the aim of these courses is to learn programming, which is commonly translated into the ability to produce code using language constructs. Writing programs is, however, only one part of programming skills; the ability to read and understand code is also essential, particularly since a programmer spends much time maintaining code written by somebody else.

One could assume that students learning to write programs automatically also learn how to read and trace code. Research has, however, indicated otherwise. For instance, Winslow (1996) notes that ”[s]tudies have shown that there is very little correspondence between the ability to write a program and the ability to read one. Both need to be taught [...]” (p. 21). In 2004, a working group at the ITiCSE conference (Lister *et al.*, 2004) tested students from seven countries on their ability to read, trace and understand short pieces of code. The results showed that many students were weak at these tasks.

Why then is it difficult to read code? Spinellis (2003) makes the following analogy: ”when we write code, we can follow many different implementation paths to arrive at our program’s specification, gradually narrowing our alternatives, thereby narrowing our choices [...] On the other hand, when we read code, each different way that we interpret a statement or structure opens many new interpretations for the rest of the code, yet only one path along this tree is the correct interpretation” (p. 85–86).

Fix *et al.* (1993) cite Letovsky, who has stated that it usually is quite easy to infer the overall goals of a program only by reading the code, based on for instance variable names, comments and other documentation. Letovsky also suggests that the same thing applies to the program implementation: a person reading through a program understands the actions of each line of code separately. The difficulty arises when trying to map high-level goals to their representation in the code. Similarly, Pennington (1987a) has found that whereas experts infer what a program does from the code, less experienced programmers make speculative guesses based on superficial information such as variable names without confirming their guesses in any way. A study conducted by Lister *et al.* (2006) based on the SOLO (Structure of the Observed Learning Outcomes) taxonomy also supports these findings. The SOLO taxonomy was originally introduced by Biggs and Collis (1982), and can be used to set learning objectives for particular stages of learning or to report on learning outcomes. The taxonomy is a general theory, not specifically designed to be used in a programming context. Lister *et al.*, however, used the taxonomy to describe how code is understood by novice programmers, and illustrate the five SOLO levels applied to novice programming as follows:

Prestructural "In terms of reading and understanding a small piece of code, a student who gives a prestructural response is manifesting either a significant misconception of programming, or is using a preconception that is irrelevant to programming. For example, [...] a student who confused a position in an array and the contents of that position (i.e., a misconception)" (p. 119).

Unistructural "[...] the student manifests a correct grasp of some but not all aspects of the problem. When a student has a partial understanding, the student makes [...] an 'educated guess"' (p. 119).

Multistructural "[...] the student manifests an understanding of all parts of the problem, but does not manifest an awareness of the relationships between these parts – the student fails to see the forest for the trees" (p. 119). For example, a student may hand execute code and arrive at a final value for a given variable, still not understanding what the code does.

Relational "[...] the student integrates the parts of the problem into a coherent structure, and uses that structure to solve the task – the student sees the forest" (p. 119). For instance, after thoroughly examining the code, a student may infer what it does – with no need for hand execution. Lister *et al.* also note that many of the relational responses start out as multistructural, with the student hand tracing the code for a while, then understanding the idea, and writing down the answer without finishing the trace.

Extended Abstract At the highest SOLO level, "the student response goes beyond the immediate problem to be solved, and links the problem to a broader context" (p. 120). For example, the student might comment on possible restrictions or prerequisites, which must be fulfilled for the code to work orderly.

Lister *et al.* found that a majority of students describe program code line by line, i.e., in a multistructural way, and that weak students in particular seem to have difficulties in abstracting the overall workings of a program from the code. They talk about students failing to "see the forest for the trees" (p. 119) and argue that students who are not able to read and describe programming code relationally do not possess the skills needed to produce similar code on their own.

Pennington (1987a) has developed a model describing program comprehension, according to which a programmer constructs two mental models when reading code. First, the programmer develops a program model, which is a low-level abstraction based on control flow relationships (e.g., loops or conditionals). This program-level representation is formed at an early stage and inferred from the structure of the program code. After that, the programmer develops a domain model, which is a higher-level abstraction based on data flow containing main functionality and the information needed to understand what the program truly does. Similarly, Corritore and Wiedenbeck (1991) have found that novices tend to have concrete mental representations of programming code (operations and control flow) with only little domain-level knowledge (function and data flow).

3. The Study

3.1. *Data*

The data analyzed in this study were collected during two high school introductory programming courses in 2005/2006. The majority of the 25 students had no previous programming background. The courses were taught using Python and covered the basics of imperative programming. To validate our results, as well as to deepen and widen our understanding of how novices' progress in an introductory programming course, we used data triangulation (Mathison, 1988) to investigate the same phenomenon from different perspectives. The data collection was conducted using surveys, progress reports and a final exam.

- The ***pre-course survey*** was used to collect background information about the students, and, e.g., their programming experience, whereas the ***post-course survey*** provided information about how the students had experienced the course.
- A ***progress report*** can be seen as a type of learning diary written on paper, aiming at revealing the students' own opinions and thoughts about their learning. What differentiates it from a traditional diary is that in addition to calling for self reflection, the report makes it possible for the teacher to evaluate students' understanding based on their responses to "trace and explain" questions. In this study, we used two progress reports that were handed out after 1/3 and 2/3 of the course respectively. Each report included a piece of code dealing with topics recently covered in the course as well as four questions. First, in the "trace and explain" questions, the students were to read the code and in their own words (1) describe what each line of the code does, and (2) explain what the program as a whole does on a given set of input data. In addition, students were asked to state what they had learned and what had been most difficult so far in the course.
- To gain further insight into the individual progress and to see how well students were able to write code related to the topics dealt with in the progress reports, we analyzed code they generated on the ***final exam***. We selected a sample of 10 exams, for which we analyzed the students' answers to two assignments: 1) a "trace and explain" question similar to the ones in the progress reports and 2) a programming task that involved developing a function that calculates and returns either the factorial of a given number or the mean of a list of numbers. Care was taken when choosing the exams to ensure that they would be representative for all students, and include work of students at different skill levels.

In total, we have analyzed 50 progress reports (two for each of the 25 students), 25 post-course surveys and 10 exams. Given that we knew who had written each answer, we we were able to analyze the progress on an individual basis.

3.2. *Method*

The progress reports, exam assignments and post-course surveys were analyzed in order to investigate three questions: how do students (1) understand program code as a whole,

(2) understand individual constructs, and (3) perceive the difficulty level of different programming topics.

The first two questions were addressed by grouping the explanations given for the "trace and explain" questions according to qualitative differences found in the data. On this point, the study resembles the SOLO study by Lister *et al.* (2006) to some extent. However, in this study, we explicitly asked students for both a multistructural and a relational response for each piece of code, giving us the possibility to compare how well the two responses match for individual students. Using data collected on the final exam, we were also able to analyze to what extent the difficulties to understand code experienced during the course were still an issue at the end of the course.

Finally, to address the third question, we studied the difficulty level of topics as perceived by the students by looking at both the progress reports and the post-course survey. The "trace and explain" questions in the reports and the final exam also provided data pertinent to this part of the study as explanation errors were considered indications of student experiencing problems with that specific topic.

4. Results

4.1. *Program Understanding*

The "trace and explain" code given in the first progress report is listed below (Algorithm 1).

Algorithm 1. Program given in progress report 1

```
try:
    a = input("Input number: ")
    b = input("Input another number: ")
    a = b

    if a == b:
        print "The if-part is executed..."

    else:
        print "The else-part is executed..."

except:
    print "You did not input a number!"
```

Students were asked to explain what this piece of code does if two integers are given as input. The analysis of the overall explanations gave rise to four categories:

1. Correct explanation ($n = 13$).
2. Choosing the wrong branch in the selection after not explaining the meaning of the statement `a = b` ($n = 5$).

3. Choosing the wrong branch although having explicitly explained the statement `a = b` ($n = 3$).
4. Giving an output totally different from the ones possible based on the code ($n = 4$).

The first category is straight forward: over half of the students gave a perfect overall explanation for the program code, not only stating the output but also explaining why that specific output was produced. The second category covers responses in which the student had failed to explain what the `a = b` statement means, and hence also missed the fact that when arriving at the selection statement, `a` does, in fact, equal `b`.

In the third category, students who explicitly explained the `a = b` statement still believed that the else-branch would be executed. This indicates a misunderstanding related to either the effects of an assignment statement, or the workings of the selection statement. In the fourth category, students appeared to be guessing, thinking that the program would output something that might have been expected from the code (e.g., values instead of one of the text messages) but that nevertheless was incorrect.

Algorithm 2. Program given in progress report 2

```
def divisible(x):
    if x % 2 == 0:
        return True
    else:
        return False

default = 5
number = default

while number > 0:
    try:
        number = input ("Give me a number: ")
        result = divisible(number)
        print result
    except:
        print "Numbers only, please!"
```

Algorithm 2 shows the piece of code included in the second progress report. For this program, students were given a list of input data including positive integers and a character, ending with a negative integer. The explanations were analyzed in a similar manner to the corresponding task in the first progress report, and four categories were found:

1. Correct explanation ($n = 8$).
2. Not understanding what it means to return a boolean value ($n = 10$).
3. Missing the last iteration, otherwise correct ($n = 4$).
4. Incorrect output ($n = 3$).

The number of correct overall explanations for the program in the second progress report was smaller than for the first report. The main stumbling block was related to understanding what happens when a subroutine returns a boolean value. The program checks if the input is divisible by two and outputs either `True` or `False` based on the result as long as the input number is positive. Some students thought that returning `True` results in the control being returned to the main program, whereas returning `False` makes the subroutine start all over again. Another misunderstanding was that there is no output whenever the subroutine returns `False`.

The third category indicates that some students also had difficulties deciding when a while loop stops, missing the last iteration (the loop will be executed once more after a negative value is input). The fourth category is similar to the one for the first progress report, i.e., students seemed to be guessing, stating that the program outputs something totally different (in this case the input values) from what the subroutine returns.

4.2. *Understanding of Individual Statements*

In order to further analyze students' skills to read and understand code, we analyzed how they explained individual statements related to a set of programming topics. The explanations found were categorized as one of the following types:

- *Correct* – the student explained the statement "by the book".
- *Missing* – the student did not write any explanation for that given statement.
- *Incomplete* – the student's explanation was correct to some extent, but lacked some parts.
- *Erroneous* – the student gave an incorrect explanation.

Although the number of students giving correct overall explanations for the programs decreased from the first to the second report (as discussed in Section 4.1), the results presented in the diagram in Fig. 1 indicate that at the same time the students became better at understanding individual statements: the number of correct explanations for individual statements increased while the number of incomplete or missing explanations decreased. When comparing the report results for each student separately, a positive progress trend

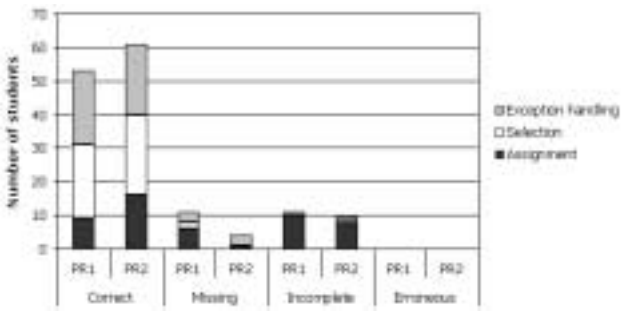

Fig. 1. The frequency of different types of explanations given by students for individual statements in the two progress reports.

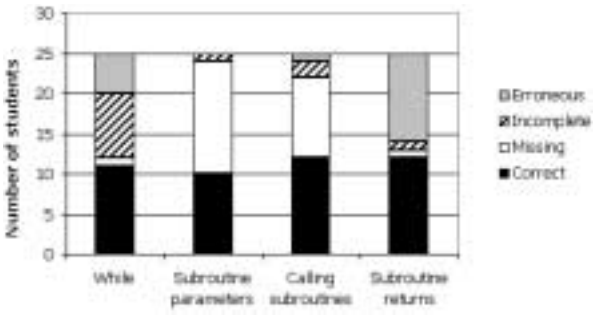

Fig. 2. The frequency of different types of explanations given by students for new topics in the second progress report.

was found: e.g., for assignment statements, 14 students improved their ”type” of explanation (e.g., from missing or incomplete to correct).

We found no erroneous comments related to the topics covered in both reports. The second progress report, however, introduced some new topics not present in the first one. The distribution of explanation types for these is illustrated in Fig. 2. The data in the diagram reflect the previously mentioned difficulties related to subroutines returning boolean values: almost half of the students gave incorrect explanations on this point. Interestingly, the other half of the students explained the returns correctly. Many students did not explain the other aspects of subroutines, such as calls or parameters, which makes it difficult to say anything about how well these topics were understood.

4.3. *Difficulty of Topics*

Apparently, assignment statements constituted the most common difficulty for students in the first progress report. However, this was not reflected in the students' own opinions on what they found difficult in the course at that time. Instead, they mentioned topics such as loops ($n = 4$), the selection statement ($n = 2$) and lists ($n = 3$). Moreover, 40% of the students only gave an incomplete explanation for what `a = b` means, not mentioning values or variables, but stating for instance that ”a becomes b” or ”a is b”.

In the second progress report, the problems students faced in the ”trace and explain” questions (returns and subroutines) were in line with the difficulties they reflected upon in the other questions: almost half of the students stated that subroutines were most difficult. Some students still reported having problems with lists ($n = 2$) and loops ($n = 2$).

In the post course survey, students were asked to rate each course topic on the scale 1–5 (1 = very easy, 5 = very difficult). The results showed that the perceived difficulty levels were quite consistent with the corresponding results presented in our previous study (Grandell *et al.*, 2006). Subroutines, modules, files and documentation were still regarded as most problematic (average difficulty of 2.8–3.2). There was, however, one exception: in the previous study, exception handling was also experienced as one of the most difficult topics (average of 2.9), but in the current study this was no longer the case (average of 2.1). The progress reports supported this finding: exception handling was not mentioned

as a difficult topic at all, and nearly all students gave a perfect explanation for statements dealing with exception handling in the ”trace and explain” questions.

When analyzing the exam assignments, focus was put on the use of assignments, if-statements, exception handling, loops and subroutines (declaration, calling and returns), i.e., the topics covered in the progress reports. Any difficulties found related to other topics were, naturally, also taken into account. As a whole, the analysis of the student solutions to the programming assignment indicated that the students, without problems, had implemented the topics that they had found difficult to explain in the progress reports. All students but two had clearly understood the problem and developed an algorithm to solve it. Half of the students had written perfect programs, whereas two of them seemed to have had problems with lists and inputting multiple values. In addition, the ”trace and explain” question included in the exam showed that the students were also able to explain these topics without difficulty. It thus seems as if students had overcome the difficulties they had had earlier in the course.

5. Discussion

The ”trace and explain” questions in the progress reports revealed some surprising results. For instance, we had not expected assignment statements to be difficult. Nevertheless, this was one of the main problems in the first progress report. Explaining `a = b` with something like ”a equals b” or ”a becomes b” is not a valid explanation; clearly, such a student has some idea of what happens, but without mentioning values the explanations are not exact enough. According to the students themselves, assignment statements were, however, not a problem. This strengthens our previous findings (e.g., in (Grandell *et al.*, 2006)): novices do not always recognize their own weaknesses and are therefore not able to point them out.

Moreover, we had expected that subroutines would be perceived as difficult in the second report, since these are commonly one of the main stumbling blocks in introductory programming (Grandell *et al.*, 2006; Haataja *et al.*, 2006; Lahtinen *et al.*, 2005). However, the analysis revealed that this was not necessarily the case; instead of subroutines per se being the problem, surprisingly many students seemed to not understand the effects of a subroutine returning a boolean value.

Many students did not explain subroutine calls or parameters explicitly, and it is thus impossible to say anything definite about how well the students understood those topics. If all missing explanations indicate ”erroneous explanations”, the number of students not understanding subroutine calls and parameters is alarmingly high. On the other hand, if the missing explanations were due to students finding those aspects ”obvious”, and therefore not needing any explanation, the number of correct explanations for those topics would be high. When taking into account that the overall explanations to the code in the progress reports did not indicate any specific difficulties in calling subroutines with parameters, the latter explanation might be a bit closer at hand.

Given the obvious difficulties with at least parts of the subroutine concept in the progress reports, we had expected to find some problems related to subroutines in the

exam. However, our analysis showed that all students had defined and used their own subroutines in an exemplary way in the programming assignment. In our opinion, the difficulties that novices encounter with the different aspects of subroutines nevertheless merit further investigation. Doing so, instruction could start focusing more on the problematic aspects, not on the ones that have traditionally been given most attention (unless these are the ones found to be the main difficulties).

We were pleased to see that exception handling was no longer among the most difficult topics, as we had made changes to the syllabus in order to facilitate students' learning of this particular topic. We now introduced exception handling at the very beginning of the course together with variables, output and user input, and students got used to check for and deal with errors from the start. It thus seems as if the order in which topics are introduced does have an impact on the perceived difficulty level, as suggested by Petre *et al.* (2003), who have found indications of topics being introduced early in a course to be perceived as ”easy” by students, whereas later topics usually are considered more difficult. One could thus expect that the difficulty level of other ”difficult” topics could be decreased by introducing the topic earlier in the course. Naturally, all topics cannot be moved to the beginning of the course just to make them all easier for the students – it is up to the teacher to decide what topics he or she thinks are most important, and then consider introducing them early in the course.

Having analyzed the explanations for both individual statements and entire programs, we can conclude that more students were able to correctly explain the program line by line than as a whole. This was found for both progress reports. When related to the levels in the SOLO taxonomy, most students were able to give correct explanations in multistructural terms, but only part of them did so relationally. Moreover, the analysis showed that students' ability to explain given individual topics increased from the first to the second progress report. This can, however, be seen as quite natural as one could – and should – expect students to gain a better understanding for individual topics as the course goes on and they become more experienced and familiar with the topics.

The categories found were quite similar for both progress reports, and can be related to the SOLO taxonomy as presented by Lister *et al.* (2006). The first category (correct explanation) contains relational responses, whereas the second and third ones (indicating a misunderstanding) can be seen as containing explanations at the pre- or unistructural level. The fourth category (guessing) includes unistructural responses, which can also be seen as the ”speculative guesses” mentioned by Pennington (1987a).

In order to bring further light on the ”guessing” one could consider conducting additional interviews with some students. By talking to the students about their answers, one could remove some of the speculations and get a better understanding for different answers. Were the students only guessing, or do the answers originate in some subtle misunderstanding that needs to be corrected? Another more resource light approach would be to let the students evaluate (e.g., on a given scale) how confident they are about their explanations. This would make it easier to distinguish between students truly believing in their answers and those merely guessing.

The difficulties found in the student generated code on the exam were not the ones found in the ”trace and explain” questions in the progress reports. Rather, the problems

found in the students' programs were better in line with what they had mentioned as difficult (e.g., lists) in the corresponding questions in the progress reports. It thus seems as when students are asked about what they find difficult about programming, they mainly answer based on their experiences from writing code – not reading it. This is, however, a natural result as this is also the main focus of instruction.

6. Conclusion

There seems to be a general lack of attention to program comprehension skills in education. Understanding the workings of an algorithm requires time and practice, and if students cannot understand the code presented they invent their own conceptions and strategies. This is certainly something we want to minimize and avoid. The study reported on in this paper illustrates an approach for teachers to evaluate student progress throughout a course and not merely based on a final exam. Our initial experience from using this approach is positive, especially with regard to the progress reports, as we feel that they provide important information during the course that most likely would remain uncovered otherwise.

The results from the SOLO study presented by Lister *et al.* (2006) are interesting as they divide student responses into different SOLO categories. However, asking the students for both a multistructural and a relational response makes the data even more interesting, since it gives us two different responses for each program. These can be used to analyze how well the responses match for an individual student. As seen in the previous section, students were in general able to give perfect descriptions of the programs line by line, but only a fraction of these gave a perfect explanation of what the program did as a whole. This finding suggests that novice programmers tend to understand concepts in isolation, and is thus consistent with the results presented by Lister *et al.* (2006) and with Pennington's idea of program vs. domain models (Pennington, 1987b). Our study also strengthens our previous findings related to students' awareness of their own stumbling blocks: novice programmers cannot necessarily point out their own weaknesses. In this study, another aspect of the awareness issue was raised as we found that novices seem to think about their difficulties related to programming mainly in terms of writing code, not reading it.

The progress reports serve as a simple mechanism to put a stronger focus on the need for also being able to read and understand code. In this study, the progress reports were mainly used as a feedback tool to perform *continuous checkups* of student progress throughout the course. They can also be used as a basis for intervention during the course, for instance in the form of *individual discussions* in which the teacher would review the progress reports and sort out potential difficulties with each student separately. The extra resources needed (teacher/tutor effort and time) might not be available, and a less demanding alternative would be to only arrange discussions with students who based on the report are in evident need of help.

As educators, we expect students to go through and learn from examples when we introduce a new topic. Doing so, the student's attention is on the construct (program

model) and not on understanding how the given piece of code solves a particular problem (domain model). The progress reports can be used as a *evaluation tool* for evaluating how well our students are doing on the domain level, and give us indications about topics that need to be further explained or taught in another way. We believe that, if used wisely, the information gathered in the reports can make a big difference for both us as educators and our students learning to program.

Acknowledgements Special thanks to Mia Peltomäki and Ville Lukka for collecting the data.

References

Biggs, J., and K. Collis (1982). *Evaluating the Quality of Learning – the SOLO Taxonomy*. New York, Academic Press.

Corritore, C., and S. Wiedenbeck (1991). What do novices learn during program comprehension. *International Journal of Human-Computer Interaction*, **3**(2), 199–208.

Fix, V., S. Wiedenbeck and J. Scholtz (1993). Mental representations of programs by novices and experts. In *INTERCHI '93: Proceedings of the INTERCHI '93 Conference on Human Factors in Computing Systems*. Amsterdam, The Netherlands, IOS Press. pp. 74–79.

Grandell, L., M. Peltomaki, R.-J. Back and T. Salakoski (2006). Why complicate things? Introducing programming in high school using python. In D. Tolhurst and S. Mann (Eds.), *Eighth Australasian Computing Education Conference (ACE2006)*, CRPIT, Hobart, Australia.

Haataja, A., J. Suhonen, E. Sutinen and S. Torvinen (2001). High School Students Learning Computer Science over the Web. *Interactive Multimedia Electronic Journal of Computer-Enhanced Learning*. Available online: `http://imej.wfu.edu/articles/2001/2/04/index.asp`. Retrieved August 29, 2006.

Lahtinen, E., K. Ala-Mutka and H.-M. Järvinen (2005). A study of the diffculties of novice programmers. In *ITICSE '05: Proceedings of the 10th Annual ITiCSE Conference*. Capacrica, Portugal, ACM Press. pp. 14–18.

Lister, R., E. S. Adams, S. Fitzgerald, W. Fone, J. Hamer, M. Lindholm, R. McCartney, J. E. Moström, K. Sanders, O. Seppälä, B. Simon and L. Thomas (2004). A multi-national study of reading and tracing skills in novice programmers. *SIGCSE Bull.*, **36**(4), 119–150.

Lister, R., B. Simon, E. Thompson, J. L. Whalley and C. Prasad (2006). Not seeing the forest for the trees: novice programmers and the SOLO taxonomy. *SIGCSE Bull.*, **38**(3), 118–122.

Mathison, S. (1988). Why Triangulate? *Educational Researcher*, **17**(2), 13–17.

Pennington, N. (1987a). Comprehension strategies in programming. In *Empirical Studies of Programmers: Second Workshop*. pp. 100–113.

Pennington, N. (1987b). Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, **19**(3), 295–341.

Petre, M., S. Fincher, J. Tenenberg *et al.* (2003). "My Criterion is: Is it a Boolean?": A card-sort elicitation of students' knowledge of programming constructs. *Technical Report 6-03*. Computing Laboratory, University of Kent, UK.

Spinellis, D. (2003). Reading, writing, and code. *ACM Queue*, **1**(7), 84–89.

Winslow, L.E. (1996). Programming pedagogy, a psychological overview. *SIGCSE Bull.*, **28**(3), 17–22.

L. Mannila is a PhD student at the Department of Information Technologies at Åbo Akademi University and at Turku Centre for Computer Science. Her main research interests are alternative approaches to teaching introductory programming. She is also interested in e-learning and leads a project at the university, within which non-university students are offered basic university level computer science courses on the web.

Pradedančiųjų pažanga mokantis programavimo pradmenų kurso

Linda MANILA

Straipsnyje pristatoma mokytojams skirta priemonė besimokančiųjų pažangai vertinti. Ji tinka ne tik galutiniam vertinimui gauti, bet ir vertinti pažangą per visą mokymosi procesą. Pristatomos pažangos proceso ataskaitos, aprašoma, kaip tai gali būti panaudota vertinant besimokančiųjų programavimo mokymąsi. Kartu su duomenimis iš apklausų ir egzaminų pažangos proceso ataskaitos gali būti naudojamos sukonstruoti išsamų besimokančiojo mokymosi eigos aprašymą ir gali pagelbėti atsakyti į šiuos klausimus: 1) kaip besimokantieji supranta programos tekstą; 2) kaip besimokantieji supranta įvairias konstrukcijas; 3) suvokti įvairių programavimo temų sudėtingumą. Taip pat pristatome rezultatus, gautus, naudojantis šia priemone vidurinės mokyklos programavimo pradmenų kursui. Pirmoji patirtis, įgyta, naudojantis pažangos proceso ataskaitomis, yra teigiama, reikšmingos informacijos pateikiama kurso teikimo metu.