Questions About Learners' Code: Extending Automated Assessment Towards Program Comprehension

Teemu Lehtinen
Questions About Learners' Code: Extending Automated Assessment Towards Program Comprehension

Teemu Lehtinen

A doctoral thesis completed for the degree of Doctor of Science (Technology) to be defended, with the permission of the Aalto University School of Science, at a public examination held at the lecture hall T2 of the school on 17 May 2024 at 12.

Aalto University
School of Science
Department of Computer Science
Learning + Technology Research Group (LeTech)
Abstract

Novice programmers have a limited understanding of the program code they produce. Their programs are often based on code snippets from examples and internet searches. Recently and rather suddenly, artificial intelligence has changed programming environments that can now suggest and complete entire programs based on the available context. However, the ability to comprehend and discuss programs is essential in becoming a programmer who is responsible for their work and can reliably solve problems as a member of a team.

Many introductory programming courses have hundreds of students per teacher. Therefore, automated systems are often used to produce immediate feedback and assessment for programming exercises. Current systems focus on the created program and its requirements. Unfortunately, their feedback helps students in iterating toward acceptable code rather than acquiring a deep understanding of the program. This dissertation addresses that gap. The dissertation defines and introduces questions about learners' code (QLCs). After a student has submitted a program, they are asked automated, personal QLCs about the structure and the logic of their program. The dissertation describes a system to generate QLCs and contributes three open-source implementations supporting Java, JavaScript, and Python.

The empirical contributions of the dissertation are based on multiple studies that research both quantitatively and qualitatively how novice programmers answer various types of QLCs. From the students, who create a correct program, as many as 20% may answer incorrectly about concepts that are critical to systematically reason about their program code. More than half of the students fail to mentally trace the execution of their program. This confirms that novices' program comprehension needs improvement and instructors may overestimate their abilities. The more students answer incorrectly to QLCs, the more they tinker with their code and have less success on the course. Current artificial intelligence systems respond to QLCs better than the average novice. However, they also lapse into humanlike errors producing failed reasoning about the code they generated, which could present an important learning opportunity for the critical use of AI in programming.
Tekijä
Teemu Lehtinen

Väitöskirjan nimi
Kysymyksiä oppijan ohjelmakoodista: Automaattisen arvioinnin kehittäminen ohjelmien ymmärtämisen tueksi

Julkaisija Perustieteiden korkeakoulu

Yksikkö Tietotekniikan laitos

Sarja Aalto University publication series DOCTORAL THESES 78/2024

Tutkimusala Tietotekniikan opetusututkimus

Käsikirjoituksen pvm 19.09.2023

Väittelyluvan myöntämispäivä 22.03.2024

Kieli Englanti

Monografia

Artikkeliväitöskirja

Esseeväitöskirja

Tiivistelmä
Aloitteellina ohjelmoinnissa on puutteellinen ymmärrys ohjelmakoodista, jota he tuottavat. Heidän ohjelmanssa perustuvat usein esimerkkeistä ja internethuista poimittuihin koodin pætkiin. Hiljattain ja melko yllättäen tekoäly on muuttanut ohjelmointi- ja päättäjän taidon. Tätä aiheuttavaa järjestelmä kontrolloi ja kauan huhuttelevat kokonaisia ohjelmia asiantuntijoiden perusteella. Ohjelmoinnissa tulee kuitenkin kantaa vastuuta omissa tuloksissa ja oman ymmärtämisen ja niistä keskeistä perusteella heille välttämättömiä taitoja.


Avainsanat

ohjelmoinnin alkeisopetus, automaattinen arviointi, tuottamaton menestys, ohjelman ymmärtäminen, hauras tieto, metakognitio

ISBN (painettu) 978-952-64-1767-7

ISBN (pdf) 978-952-64-1768-4

ISSN (painettu) 1799-4934

ISSN (pdf) 1799-4942

Julkaisupaikka Helsinki

Painopaikka Helsinki

Vuosi 2024

Sivumäärä 220

Preface

The path that led me to write this dissertation had many curves and at times disappeared from my view. Without the supporting and inspiring discussions with Professor Lauri Malmi and my advisor Ari Korhonen, I would have lost on the way. Thank you for your trust, patience, advice, and fellowship over all the years.

Ten years ago, Otto Seppälä and Juha Sorva hired me to Learning + Technology research group as a summer intern. After this turning point, I have continuously received the necessary support to complete several degrees and the opportunity to learn new board games. Years later, I am proud of writing articles with both Otto and Juha. I hope to have learned a thing or two from their example in both teaching and research. In the first autumn after my internship, I joined Teemu Sirkiä, Lassi Haaranen, and Aleksi Lukkarinen in room B207. Their company and example were irreplaceable to start me on my way. I must thank Lassi for always putting his neck out for our ideas and making me write a research article before I knew how.

This dissertation started to take form after I met with André Santos who offered new influence and clear thinking to formulate the thoughts. Thank you for writing the articles with me and for your continued support. During my doctoral studies, I have met too many first-class colleagues to list all here. You have supported my work through your interest and offered a heartwarming community here at Aalto, abroad at conferences, and virtually over the Internet. I want to mention our working group for the conference in Dublin and thank them for the intensive, yet uplifting experience. From our research group, I am glad to share articles also with Sami, Juho, Charlie, and Arto after admiring their work for the past years.

My preliminary examiners, Professor J. Ángel Velázquez-Iturbide and Professor Petri Ihantola provided encouragement and perceptive obser-
vations that have truly improved this dissertation. Thank you for all your trouble. I am honored to have Professor Mikko-Jussi Laakso as my opponent.

All my work is built on the unconditional love and support of my family. Through all the different ups and downs I could always rely on my parents and siblings. Considering the contents of my dissertation, I learned programming from my brother and my sister is an excellent teacher. Frankly, either of them would be better prepared to pursue a doctoral degree than I have been. During my doctoral studies, my family has grown through marriage and a son. We have received help from my parents-in-law or my parents whenever there were too many tasks for us to handle. They have all offered considerable support and encouragement for my work.

Noora, although your name only appears in the preface of this dissertation I will always consider it as our team effort. You have always been there to listen and ask the most important questions when I have been sleepless, scared, or speechless. I hope this milestone could, perhaps in imperceptible ways, encourage the next generations to pursue their goals whether they be of academic or other nature. I will always offer my little experience in answer to their wisdom of asking questions.

Helsinki, March 26, 2024,

Teemu Lehtinen
## Contents

1. Preface
2. Contents
3. List of Publications
4. Author's Contribution
5. List of Figures
6. List of Tables
7. Terms and Abbreviations

### 1. Introduction

#### 2. Background

- 2.1 Types of Programming Exercises
- 2.2 Fragile Knowledge in Programming
- 2.3 Program Comprehension
- 2.4 Metacognition and Self-Regulated Learning
- 2.5 Automated Feedback Generation
- 2.6 Automated Question Generation

### 3. Questions About Learners' Code (QLCs)

- 3.1 Definition
- 3.2 Potential Benefits and Challenges
- 3.3 Design Principles
- 3.4 System Design
- 3.5 Question Design
- 3.6 Available Systems
List of Publications

This thesis consists of an overview and of the following publications which are referred to in the text by their Roman numerals.


V Teemu Lehtinen, Lassi Haaranen, Juho Leinonen. Automated Questionnaires About Students’ JavaScript Programs: Towards Gauging


VII Teemu Lehtinen, Charles Koutcheme, Arto Hellas. Let’s Ask AI About Their Programs: Exploring ChatGPT’s Answers To Program Comprehension Questions. Accepted for publication in Proceedings of the 46th International Conference on Software Engineering (ICSE ’24), Lisbon, Portugal, DOI:10.1145/3639474.3640058, April 2024.
Author’s Contribution

Publication I: “Let’s Ask Students About Their Programs, Automatically”

I am the main author of the paper and was the driving force in developing the idea as well as in writing the text with the other authors. Santos contributed most of the section on system design. Sorva clarified and considerably extended the text. I presented the paper.

Publication II: “Students Struggle to Explain Their Own Program Code”

I am the main author of the paper. I designed the study and collected the data in collaboration with Haaranen. Haaranen wrote the context description. Lukkarinen analyzed one of the three tasks and wrote the related text. I did the rest of the analysis and wrote 75 percent of the text as well as presented the paper.

Publication III: “Jask: Generation of Questions About Learners’ Code in Java”

Santos, Soares, and Garrido designed the study and collected the data. Soares implemented most of the program code. Santos is the main author and wrote the majority of the text. I analyzed the data and wrote the results, which is about 20 percent of the text, and presented the paper.
Publication IV: “Towards Giving Timely Formative Feedback and Hints to Novice Programmers”

My role in the working group was to investigate, process, and augment datasets on programming processes with Sarsa, Petersen, and Jeuring. I implemented program code, participated in coding data, prepared augmented data for publication, and wrote descriptions of datasets and tools in Section 4. Other authors were responsible for other parts of the work.

Publication V: “Automated Questionnaires About Students’ JavaScript Programs: Towards Gauging Novice Programming Processes”

I am the main author of the paper and wrote about 90 percent of the text. I designed the study with Haaranen. I implemented the program code, collected the data, and did the analysis. Haaranen and Leinonen helped to focus and clarify the paper. Leinonen presented it.

Publication VI: “Automated Questions About Learners’ Own Code Help to Detect Fragile Prerequisite Knowledge”

I am the main author of the paper and wrote about 80 percent of the text. I designed the study, implemented the program code, and presented the paper. Korhonen helped to collect the data and discuss the analysis. Seppälä improved focus and clarified all sections in the paper.

Publication VII: “Let’s Ask AI About Their Programs: Exploring ChatGPT’s Answers To Program Comprehension Questions”

The two first authors collaborated to design the study and analyze the data. I collected the data and wrote about 50 percent of the text. Hellas improved, clarified, and expanded the paper and contributed most of the introduction, background, and conclusion sections.
List of Figures

2.1 An example of a step in traversing an AST using DFS and a stack ................. 31

3.1 An example set of possible questions about a variable in the program. ............... 38

3.2 A simplified design for a template-based QLC system from PI. ......................... 45

3.3 A flowchart describing an algorithm to detect six different roles of variables. .......... 48

3.4 Using QLCJS in a programming task in PV. ............... 52

6.1 Course success after three programming exercises from PII for students who failed, passed only functional tests, or passed both tests and QLCs. ......................... 70

6.2 The distribution of students’ average number of errors while answering QLCs and correlation with the total points they received in PV. ......................... 71

6.3 Course point distributions for each QLC type and whether students answered correctly or incorrectly from PVI. ............... 72

6.4 The six students described qualitatively in PV. ............... 76

7.1 Success rates by QLC type and programming task for the two LLMs from PVII. ......................... 82

7.2 Number of incorrect answers by QLC type and error code for the two LLMs from PVII. ......................... 84

8.1 Students’ success rates for each QLC type and researched programming task. ......................... 89

8.2 Example A—Student’s work on the assignment in PVI ............... 94

8.3 Example B—Student’s work on the assignment in PVI ............... 95
8.4 Example C—Excerpt from student's work on the assignment in \textbf{PVI} \hfill 96
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Chapters of the dissertation and their relations to publications.</td>
<td>17</td>
</tr>
<tr>
<td>2.1</td>
<td>The taxonomy of task types summarized from [12].</td>
<td>20</td>
</tr>
<tr>
<td>2.2</td>
<td>Types of fragile knowledge Perkins and Martin [82] identify in programming.</td>
<td>21</td>
</tr>
<tr>
<td>2.3</td>
<td>Roles of variables paraphrased from [98].</td>
<td>23</td>
</tr>
<tr>
<td>2.4</td>
<td>Block Model of program comprehension paraphrased from PI and [103].</td>
<td>25</td>
</tr>
<tr>
<td>2.5</td>
<td>Program explanation categories paraphrased from [138].</td>
<td>26</td>
</tr>
<tr>
<td>2.6</td>
<td>Categories of analysis techniques from two literature reviews.</td>
<td>29</td>
</tr>
<tr>
<td>3.1</td>
<td>Example QLCs from PI that target different areas of program comprehension in Block Model.</td>
<td>40</td>
</tr>
<tr>
<td>3.2</td>
<td>An example of transforming AST nodes to record the history of each variable state.</td>
<td>49</td>
</tr>
<tr>
<td>3.3</td>
<td>Available QLC generation systems.</td>
<td>51</td>
</tr>
<tr>
<td>4.1</td>
<td>The timeline of QLC studies and their main characteristics.</td>
<td>56</td>
</tr>
<tr>
<td>5.1</td>
<td>Manually prepared QLCs with the targeted program comprehension area in Block Model and students success rates in PII.</td>
<td>59</td>
</tr>
<tr>
<td>5.2</td>
<td>QLC types and students success rates on the two questionnaires in PIII.</td>
<td>60</td>
</tr>
<tr>
<td>5.3</td>
<td>QLC types and students success rates of their first attempt in PV.</td>
<td>61</td>
</tr>
<tr>
<td>5.4</td>
<td>QLC types with the targeted program comprehension area in Block Model and students' success rates in PVI.</td>
<td>62</td>
</tr>
</tbody>
</table>
5.5 The numbers of answers for each **incorrect option** in PV or PVI. ........................................ 65
5.6 Results of a survey on students’ perceptions about answering QLCs in PIII. ............................... 66
6.1 Guidelines from PIV for identifying when to intervene in students’ program creation process. .......... 74
7.1 Generated QLC types from PVII with the targeted program comprehension area in Block Model, the number of generations (N), and success rates for the two LLMs. .................. 81
7.2 The coding developed in PVII for incorrect answers from the LLMs to the QLCs. .......................... 83
7.3 Number of incorrect answers from the two LLMs for each error code from PVII. ........................... 84
8.1 All QLC types, which have been evaluated in the dissertation, and students’ success rates. ................. 88
8.2 Evaluated QLC types located to Block Model. ............... 89
Terms and Abbreviations

Abstract syntax tree (AST) A tree model of the syntactic structure of a program or some source code.

Artificial intelligence (AI) Computing systems that perform tasks, which previously required human intelligence.

Automated assessment Automated evaluation and possibly feedback of student’s work—often program code in computer science.

Block Model A model of program comprehension that is aimed toward education [103]. It includes two dimensions that represent increasing levels of abstraction.

Computing education research (CER) The study of how computing topics are learned and taught.

Concrete operational form A cognitive development stage. In this dissertation, refers to having the ability to logically infer about concrete code in concrete context.

Depth-first search (DFS) An algorithm and order to traverse elements of data structures, that are organized as tree models.

Distractor An incorrect option for a multiple choice question, that may distract from selecting the correct answer.

Integrated development environment (IDE) A software designed for software development tasks, such as editing and testing code.

Large language model (LLM) A probabilistic model, typically a neural network, that is pre-trained to generate text for prompts and can power AI systems.
**Metacognition**  Thinking about thinking, including questions such as, what you actually know, and how your learning proceeds.

**Multiple-choice question (MCQ)**  A type of question where the answer is selected among provided options.

**Program comprehension (PRC)**  Understanding program code, including its design, execution, and purpose.

**Question about learners’ code (QLC)**  A question about program code, which the respondent wrote themself.

**QLC type**  A design for a QLC, that includes logic and possible templates for generating a particular type of QLC instance when applied to a target program code.

**Self-explanation**  A cognitive process where learners generate typically internal explanations for themselves about concepts they are learning.

**Self-regulated learning**  Applying metacognitive skills to direct one’s learning, including time management, problem-solving strategy, self-explanation, and motivation.

**Tracing**  Mentally simulating, predicting, and closely tracking how program code will execute step by step.
1. Introduction

The widespread role of computing in modern society has led to a broad demand for computing education, which in turn increases the interest in computing education research (CER) [31; 5]. This dissertation belongs to that area and more specifically researches education of introductory programming.

Programming is not only about writing computer programs. In their review covering 1666 papers about the education of introductory programming since 2003 and up to 2017, Luxton-Reilly et al. [68] report a widely studied conclusion that “code-reading skills are prerequisite for problem-solving activities including code writing.” For example, success in explaining or tracing\(^1\) code indicates success in writing programs [62]. Such reading skills are described as part of program comprehension (PRC), which is another broad term for understanding the design, execution, and purpose of program code. The term and the related research have roots in software engineering but PRC has been adopted and applied to introductory programming as well [104; 43]. For example, tasks to explain program code in plain English have been used as an instrument to research students’ PRC [132].

Program comprehension facilitates meaningful discussion about programs. While the ability to comprehend code is critical for systematic program writing, the benefits of PRC skills exceed merely writing or editing code. Software development is teamwork [2] where the ability to review and discuss program code with colleagues is an essential part of creating software. The importance of PRC is further emphasized by emerging programming tools that apply artificial intelligence (AI)\(^2\) to generate program

\(^1\)Mentally simulating, predicting, and closely tracking how program code will execute step by step.

\(^2\)Computing systems that perform tasks, that previously required human intelligence.
Introduction

code. While the present generation of AI systems may increase produced code, humans are still needed to assess, refine, debug, integrate, and discuss it [130].

Writing a program does not implicate comprehension of that program. An ideal programmer systematically expands their program design using plans and conventions in their knowledge to write program code [114; 92]. They confirm, at least mentally, that any new code structures are correct before moving on. On the contrary, some novices adapt and combine program code from different sources and available hints without understanding it completely [140]. After a period of tinkering and testing, they may have desired functionality without completely comprehending their code [50]. The problem is likely aggravated by replacing human educators with automated assessment systems that focus on functional errors and fixing them. This reduces opportunities to discuss students’ program code and can make tinkering a more attractive approach.

All things considered, programming students should aim to comprehend and regularly discuss the program code they create. This is supported by the success of pair programming in education [128], where two programmers share a view to the same code, and switch roles between driver, who edits code, and navigator, who asks questions and makes suggestions. In addition, educators have asked simple questions about struggling students’ program code to help them recall necessary details and make progress [82]. Similar questions have been suggested to be automated in a program editor to test students’ knowledge [39].

This dissertation applies simple questions to extend automated assessment toward students’ program comprehension. Imagine that an educator could automate questions, such as the following, for hundreds of students and the unique program codes they create.

What is the role of variable n in your program?
What values are assigned to n when your program is executed?

Furthermore, automation could assess the answers and deliver formative feedback, which helps students to measure and develop comprehension of their program. Such an approach could help to develop students’ PRC and the important ability to discuss their programs early on. Additionally, this offers students insights into what they actually know and how their learning beyond a single program writing task proceeds. The importance of supporting such metacognitive skills needed for self-regulated
Introduction

Learning has been recognized both generally and specifically for learning programming [66].

The dual research goal of this dissertation is to develop questions that extend automated assessment towards program comprehension, and evaluate how students answer the questions in large courses. The potential benefits of the research for different stakeholders include:

1. Large cohorts of students can receive practice and guidance towards PRC and self-regulated learning in one activity.

2. Educators can identify students having PRC issues in large classes and develop early support to help them.

3. Researchers can collect structured data on students’ PRC of their own programs and study their programming knowledge.

The seven publications, which this dissertation contains are denoted as PI–PVII. Table 1.1 presents the structure of the dissertation.

Table 1.1. Chapters of the dissertation and their relations to publications.

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title and additional description</th>
<th>Important publications</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
<td>PI, PIII, PV, PVI</td>
</tr>
<tr>
<td>2</td>
<td>Background</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Questions About Learners’ Code (QLCs)</td>
<td>Definition and technical results</td>
</tr>
<tr>
<td>4</td>
<td>Research Questions</td>
<td>PV, PVI</td>
</tr>
<tr>
<td>5</td>
<td>Students’ Performance and Views on QLCs</td>
<td>PII, PIII, PV, PVI</td>
</tr>
<tr>
<td>6</td>
<td>Probing Students’ Knowledge With QLCs</td>
<td>PII, PIV, PV, PVI</td>
</tr>
<tr>
<td>7</td>
<td>Artificial Intelligence Answers QLCs</td>
<td>PVII</td>
</tr>
<tr>
<td>8</td>
<td>Discussion</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Conclusion</td>
<td></td>
</tr>
</tbody>
</table>

After the introduction, Chapter 2 discusses the related research. Chapter 3 defines an approach to query students about their programs, as described in PI, and discusses possible technical designs for it. The designs are the technical results of this dissertation, including software for Java in

3Applying metacognitive skills to direct one’s learning, including time management, problem-solving strategy, self-explanation, and motivation.
PIII, JavaScript in PV, and Python in PVI. Chapter 4 defines the research questions and design for evaluating the developed approach. Chapters 5 to 7 present the studies PII–PVII, including the research results that answer the research questions for this dissertation. Particularly, Chapter 7 researches the role of this work in the nascent era of AI-assisted programming. Chapter 8 discusses the results, limitations, and future work. Finally, Chapter 9 concludes the dissertation.
2. Background

This chapter explores related research that forms the basis for this dissertation. Section 2.1 reviews types of programming exercises. Sections 2.2 to 2.4 delve into the fragility of newly acquired programming knowledge, understanding computer programs, and students’ self-awareness during programming exercises. Following that, Sections 2.5 and 2.6 focus on technologies for analyzing student programs and automating the generation of new questions.

2.1 Types of Programming Exercises

Different types of exercises are used in learning programming. Ruf et al. [96] researched four school textbooks, two university textbooks, and worksheets from a university course in introductory programming. They analyzed all programming-specific novice tasks, that is, tasks including any program code. They classified the exercises into types based on the skills students are required to use and the format of knowledge in the exercise, including code, text, or diagram. The majority of exercises, almost 40%, ask students to write code from text or diagram descriptions. In addition, code is written to use, adjust, extend, and complete the code given in the problem. Together, these tasks make up 75% of the exercises. Exercises asking to test or trace code each constitute approximately 10% of the exercises.

Exam questions have been also analyzed and their types are similarly distributed, that is, approximately half of the exercises are code writing tasks, and 10–25% are code reading tasks. [107; 84]. Petersen et al. [84] discover challenges in evaluating which skills and concepts are used in a particular exam exercise or its cognitive complexity as the course contents have a significant role. They note that both code writing and reading tasks typically require students to simultaneously understand more than 7 or
6 concepts. Bower [12] proposes a taxonomy of task types based on the level of knowledge required. Table 2.1 summarizes the levels, which are influenced by the more generic Bloom’s taxonomy for learning objectives.

Table 2.1. The taxonomy of task types summarized from [12].

<table>
<thead>
<tr>
<th>Type</th>
<th>Description of Knowledge</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Declarative</td>
<td>Recognition and recollection</td>
</tr>
<tr>
<td>2. Comprehension</td>
<td>Understanding underlying concepts</td>
</tr>
<tr>
<td>3. Debugging</td>
<td>A process relying on levels 1 and 2</td>
</tr>
<tr>
<td>4. Prediction</td>
<td>More wholistic than on level 3</td>
</tr>
<tr>
<td>5. Provide-an-example</td>
<td>Synthesizing with less cues</td>
</tr>
<tr>
<td>6. Provide-a-model</td>
<td>Synthesizing in new context</td>
</tr>
<tr>
<td>7. Evaluate</td>
<td>Making judgements</td>
</tr>
<tr>
<td>8. Meet-a-Design-Specification</td>
<td>Levels 1–7 are relevant</td>
</tr>
<tr>
<td>9. Solve-a-problem</td>
<td>Less structured task than level 8</td>
</tr>
<tr>
<td>10. Self-reflect</td>
<td>Evaluating learning</td>
</tr>
</tbody>
</table>

### 2.2 Fragile Knowledge in Programming

Perkins and Martin [82] describe fragile knowledge as follows:

Knowledge comes across as a “you have it or you don’t” sort of thing. The student may know some things about the language, but not other things and perhaps not enough. However, common experience testifies that often a person does not simply “know” or “not know” something. Rather, the person sort of knows, has some fragments, can make some moves, has a notion, without being able to marshall enough knowledge with sufficient precision to carry a problem through to a clean solution. One might say that learners in such state have fragile knowledge. [82, p. 216]

They researched students’ knowledge in sessions where a student works on programming tasks and an instructor is following their progress. When the student faces difficulties, the instructor first intervenes by asking high-level strategic questions from the student. If the student needs more help, the instructor asks questions that include a hint. If the student still does not make progress, the instructor explains how to proceed. In the
Background

sessions, high-level questions, such that students might learn to naturally ask themselves, helped students to progress 55% of the time. When such prompt was not enough, direct hints helped only 16% of the time. In addition, Perkins and Martin [82] recorded that students accurately traced their code 50% of the time and argue that tracing ability acts as a filter to reject invalid plans for the program. Based on the sessions, they discuss different types of fragile knowledge as described in Table 2.2.

Table 2.2. Types of fragile knowledge Perkins and Martin [82] identify in programming.

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partial</td>
<td>Student does not recall specific knowledge that was introduced in their class, and which instructor asks them about.</td>
</tr>
<tr>
<td>Inert</td>
<td>Student does not recall specific knowledge before the instructor asks them about it.</td>
</tr>
<tr>
<td>Misplaced</td>
<td>Student applies specific knowledge to an inapplicable design. Often students overgeneralize and try to apply lately acquired concepts too widely.</td>
</tr>
<tr>
<td>Conglomerated</td>
<td>Student produces a construct that is a mixture of ill-fitting pieces of knowledge. Often students attempt to encode tasks in a way that is not supported in the programming language.</td>
</tr>
</tbody>
</table>

To a certain degree, theories about transfer of learning explain the different types of fragilities. In this theory, transfer refers to the cognitive task of recognizing what previous knowledge is relevant and how it can be applied in the current context, which always has some differences to contexts where knowledge was learned [35]. For inert knowledge, transfer occurs positively after additional guidance from the instructor. For misplaced or conglomerated knowledge, transfer occurs negatively, that is, in a context where the knowledge does not apply. Knowledge is conglomerated when it considerably tightly fixes incorrect concepts together. Such unfortunate constructs are also referred to as misconceptions, which have received considerable attention in CER over the years [7; 16; 27; 32; 48; 90; 111; 117; 118; 119; 123]. Knowledge on misconceptions and their nature helps to develop means for detecting, preventing, and refuting them.
2.2.1 Students’ Programming Ability After Introductory Course

McCracken et al. [75] find that students, who have completed their first year of computing courses, succeeded below instructors’ expectations in programming exercises. In this multinational and multi-institutional study, a majority of students did poorly and often struggled to decompose implementable sub-problems from exercise descriptions. Later studies have discovered that students perform better and more according to instructors’ expectations when code scaffolding and familiar development environment are provided [74]. When providing a test harness, including test runner and unit tests to verify programming results, 13% of students failed to produce any functionally correct code, and 9% of students failed to complete a more complex programming task [129]. Robins [94] argues that a fragile understanding of a programming concept makes it harder to acquire many tightly integrated, dependent concepts in the programming language and may explain why some students fall behind.

In another multinational study, Lister et al. [61] identify that many students, who have concluded their introductory programming courses, are weak in tracing the execution of a program. They estimate that the bottom 25% of students suffer from fundamental issues preventing them from tracing correctly. Moreover, some students are reluctant to trace programs. Students have described tracing as an unrealistic exercise and impossible for them to complete correctly [20; 21]. However, tracing has been considered a professional, necessary skill for developing programs since the 1980s [83; 113]. Furthermore, the ability to trace programs helps students to succeed in writing them [67; 133].

Ma et al. [69] investigated students’ mental models on value and reference assignments after completing an introductory programming course. They conclude that the proportions of students having nonviable mental models are roughly 30% for value assignment and 83% for reference assignment. Many of those mental models likely included one or more misconceptions.

In light of these findings, students, who complete their first programming course, regularly have misconceptions, trace programs poorly, and need scaffolding as both code and programming tools. In other words, a considerable portion of students have fragile knowledge of programming after their first semester.
2.3 Program Comprehension

Reading and understanding computer programs has been studied since the 1970s with a focus on professional software engineering and large code bases [37; 102; 137]. Different models of program comprehension were suggested in the 1980s [58; 81; 114]. Research in introductory programming often refers to the model by Soloway and Ehrlich [114], which unites plans from problem-solving and schemata from cognitive psychology to describe chunks of knowledge that programmers learn to recognize in program code. For example, a simple plan, which some sources can describe as schema or template, could consist of swapping variable values or searching for an element from a list. In addition to comprehending, plans have been applied to generating programs as well [23; 92; 93]. Plans may have beacons, which are code features that help programmers identify plans efficiently [13].

Considering beacons, a cognitive concept called roles of variables may in part help to identify plans. There are common patterns in how variables are initialized and updated. Based on that stereotypical use of variables, Sajaniemi [97] defines ten different roles that can describe practically every variable in novices’ programs as described in Table 2.3. The roles are based on analyzing programs and their variables in three introductory programming textbooks. The ten roles can be taught in an introductory program.

<table>
<thead>
<tr>
<th>Role</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed value</td>
<td>No values are assigned after initialization.</td>
</tr>
<tr>
<td>Stepper</td>
<td>A predictable succession of values is assigned.</td>
</tr>
<tr>
<td>Most-recent holder</td>
<td>The latest value is assigned when going through a series of values.</td>
</tr>
<tr>
<td>Most-wanted holder</td>
<td>The best value is assigned when going through a series of values.</td>
</tr>
<tr>
<td>Gatherer</td>
<td>Accumulates individual values to itself when going through a series of values.</td>
</tr>
<tr>
<td>Transformation</td>
<td>Assigned values are from the same calculation on other variables.</td>
</tr>
<tr>
<td>Follower</td>
<td>Assigned values are from another variable.</td>
</tr>
<tr>
<td>One-way flag</td>
<td>A new value is assigned only once.</td>
</tr>
<tr>
<td>Organizer</td>
<td>An array used to rearrange its elements.</td>
</tr>
<tr>
<td>Temporary</td>
<td>A value is needed for a very short time.</td>
</tr>
</tbody>
</table>
Winslow [139] discusses pedagogy for learning programming and recognizes that students who understand syntax may lack practice in code patterns, programming strategies, and debugging. Regarding debugging, they mention reading and understanding code, where students may struggle even with programs they wrote themselves. Similarly, other studies highlight how strategies to write or comprehend programs complement knowledge of syntax and tools [22; 95]. In fact, understanding the syntax and semantics of a program does not alone determine how it executes [26]. To reason about execution or to trace code students also need a mental model of an abstract computer, which is referred to as *notional machine* [15; 25; 119].

Lister [60] applies neo-Piagetian theory of cognitive development to learning programming. Briefly explained, students who are novices in programming start reasoning about programs in “preoperational form”. They learn to trace code and reason based on input and output. When they reach “concrete operational form” they can abstract the purpose of code statements in their context. For example, they can realize the purpose of a procedure from the code statements without tracing it to the end with some selected values. They can logically infer about concrete code in a concrete context. Only after developing adequate chunks of knowledge, students’ cognitive capacity develops to “formal operational form”, and allows them to hypothesize new contexts for specific program code. Such skills could be detected for example in problem-solving or debugging and testing. Instructors often lecture with the expectation that students reason at the highest, formal operational form. In my special interest, Lister [60, p. 11] writes

- “it is difficult to infer that a student has used formal operational reasoning to produce a piece of code, when the only supporting evidence is the code itself”, and

- “a process that combines quasi-random code changes and copious trial runs (a process which is decidedly not formal operational reasoning) can produce a poor but passing assignment solution”.

Schulte et al. [104] surveyed program comprehension research from the perspective of teaching programming and relate different comprehension models to *Block Model* [103], which aims to support the teaching of pro-
gramming. Table 2.4 presents two axes of Block Model that form cells or areas of comprehension. Rows resemble research on text comprehension with abstraction level increasing from atom towards macro while columns mark three dimensions of comprehension. The first dimension represents structural comprehension of raw program code. The second dimension supplements it with understanding execution, that is the notional machine. The third dimension appends inferences and domain knowledge to understand goals and purposes.

Table 2.4. Block Model of program comprehension paraphrased from PI and [103].

<table>
<thead>
<tr>
<th>4. Macro</th>
<th>entire program</th>
<th>behavior of the program</th>
<th>purpose of the program</th>
</tr>
</thead>
<tbody>
<tr>
<td>3. Relational</td>
<td>connections between “blocks”; e.g., calls</td>
<td>flow between “blocks”; e.g., call sequences</td>
<td>integration of subgoals</td>
</tr>
<tr>
<td>2. Block</td>
<td>syntactically or semantically related elements</td>
<td>behavior of a “block”</td>
<td>purpose of a “block”; program subgoal</td>
</tr>
<tr>
<td>1. Atom</td>
<td>language elements</td>
<td>behavior of elements</td>
<td>purpose of elements</td>
</tr>
</tbody>
</table>

1. Text 2. Execution 3. Function

Schulte et al. [104] state that novices’ comprehension models have holes because their knowledge is fragile. Gradually, students append knowledge to different cells of Block Model and build references and coherency between them.

2.3.1 Evaluations of Program Comprehension

Whalley et al. [138] designed nine multiple-choice questions (MCQs)\(^1\) to target cognitive categories as described in Bloom’s revised taxonomy [54]. All of the questions presented program code and required students to alternatively pick the resulting value of a variable, matching diagram, equivalent code, matching description, missing lines, or repair lines. They noticed that they easily underestimate the cognitive complexity of a question, which may be dependent on the student’s approach as well. For example, students could either mechanistically trace each code given as an option, analyze the code options in the given context, or synthesize appropriate code for the given context and compare that to the options.

\(^1\)A type of question where the answer is selected among provided options.
In addition to the MCQs, they asked students to explain a small program in free text, which they analyzed using categories they developed based on SOLO taxonomy [10]. Their categories are described in Table 2.5.

Table 2.5. Program explanation categories paraphrased from [138].

<table>
<thead>
<tr>
<th>SOLO Level</th>
<th>Code Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relational</td>
<td>Summarizes purpose of the code</td>
</tr>
<tr>
<td>Multistructural</td>
<td>Describes code line by line</td>
</tr>
<tr>
<td>Unistructural</td>
<td>Describes one portion of the code</td>
</tr>
<tr>
<td>Prestructural</td>
<td>Issues with programming structures</td>
</tr>
</tbody>
</table>

The majority of answers are on a multistructural level. Stronger students performed more at the relational level, while weaker students performed at the unistructural level at times. SOLO is based on neo-Piagetian theories discussed in Section 2.3. The SOLO levels from prestructural to relational roughly represent reasoning starting from preoperational form and ending in concrete operational form.

Other studies use similar questions and find that students’ level of reasoning predicts performance in tracing and writing code [67; 106; 133]. Students require considerable time to reach the multistructural level and may have misconceptions about sequences of statements [109]. Students’ reasoning about program code is also studied in terms of neo-Piagetian theory. Both a multiple-choice questionnaire on code examples [19] and a think-aloud study [126] indicate that at least half of the students take several weeks of study to reach the concrete operational form.

2.3.2 Pedagogical Development for Program Comprehension

Program visualization is developed and applied during introductory programming courses to support students’ mental model of a notional machine [112; 120]. Tracing is suggested to become a central concept for programming courses which should be practiced before writing code [40; 77]. Xie et al. [141] examine many findings described in this section and suggest a theory of four distinct skills: tracing, writing correct syntax, recognizing reusable templates, and using them to solve problems. In an initial evaluation, they aim to decrease cognitive demand by providing explicit and incremental instruction of the distinct skills, and according to results, improve learning. Izu et al. [43] identify, define, and classify learning ac-
activities for program comprehension. They suggest a hypothetical learning trajectory based on Block Model could help teachers design the adoption of program comprehension activities in programming education.

2.4 Metacognition and Self-Regulated Learning

Metacognition and self-regulated learning have been studied extensively for general education. A recently updated systematic literature review presents the previously applied theories in programming education and makes further recommendations [66; 88]. Next, I discuss recent programming education publications in the area of metacognition.

Loksa et al. [65] evaluated a design to promote metacognitive awareness during a programming camp. They provided explicit instruction on six stages of problem-solving in programming and prompted learners regularly to describe their current stage. According to the study, the approach successfully promotes independence, increases self-efficacy gains, and reinforces a growth mindset. The stages, which learners can revisit many times, are verbatim [65, p. 1451]:

1. Reinterpret problem prompt
2. Search for analogous problems
3. Search for solutions
4. Evaluate a potential solution
5. Implementing a solution
6. Evaluate implemented solution

In a think-aloud study, Loksa and Ko [64] investigated how self-regulation supports solving programming problems. They argue that ideal learners plan their problem-solving steps, monitor their progress at a step, monitor their comprehension of related concepts, reflect on their cognitive performance, and self-explain their decisions. In their study, most self-regulation behaviors are shallow. Among more experienced students, planning behaviors correlate with having fewer errors. Illogically, self-explanation behaviors correlate with having more errors. While self-explanation is naturally silent, facing errors in a think-aloud study may specifically prompt spoken attempts to self-explain. In fact, self-

\footnote{Self-explanation is a cognitive process where learners generate typically internal explanations for themselves about concepts they are learning [11].}
explanation prompts have been applied in computing education with positive results [56; 134] and students can be trained to self-explain better [9; 71].

Prather et al. [86] conducted a think-aloud study to research how automated assessment tools fail to support students’ metacognitive awareness. They identify incorrect interpretations or incorrect conceptual models of the exercise prompt as the most frequent issues. Furthermore, some students clung to their incorrect understanding even when they re-read the prompt. Multiple students jumped ahead in ideal problem-solving stages and ran into trouble. They did not seem to search for analogies and solutions or evaluate them before starting implementation. Finally, students interpreted feedback, including both error messages and failed test cases, as signs of being nearly done when actually their solution needed a complete redesign. Understanding of the exercise prompt can be improved by asking students to simulate or design test cases before writing code [87; 85].

2.5 Automated Feedback Generation

This dissertation aims to extend current automated feedback systems. The primary focus is on automated program analysis that can produce knowledge on the structure and behavior of the program. Such facts can be used to generate feedback, including the type of questions that this dissertation develops. Another focal point is the means to support students’ metacognition in feedback. For example, feedback could explain problem-solving strategies or pose questions that trigger self-explanation.

Two recent literature reviews provide a good understanding of the research on automated systems to support students’ programming exercises. Keuning et al. [52] systematically reviewed 146 papers about automated feedback tools since the 1960s and up to 2015. They decided to exclude a large volume of papers on similar tools that are solely based on automated program tests and their results. The authors researched the nature of generated feedback, techniques to generate feedback, adaptability of tools, and effectiveness of tools. Considering the nature of feedback, tools provide knowledge about mistakes, how to proceed, concepts, task constraints, and metacognition. The previous are in decreasing order of popularity. Only one tool provides feedback aimed to support metacognition. Vizcaíno [136] present a tool that includes an automated agent posing as a student in a chat session for a small group, which needs to agree on an answer to a
question about an example code. The agent encourages peers to explain
the correct solutions they propose and may propose incorrect solutions to
check peers’ argumentation.

Paiva et al. [79] conducted a state-of-the-art review on automated assess-
ment in computer science from 2010 up to early 2021. They researched
the extent of automated assessment in different domains of the discipline,
testing techniques, secure code execution, feedback generation techniques,
applied learning analytics, effectiveness, and expectations for the future.
Considering feedback aimed to support metacognition, they acknowledge
the idea in PI and another experiment focusing on students’ conceptual
model of the programming problem. Prather et al. [87] performed a think-
 aloud study where students started a programming exercise by reading
the prompt and answering a quiz about the expected output for a test case
before they started writing program code.

A particular interest for this dissertation is the techniques that analyze
program codes and can be applied to generate new types of feedback as
well. Table 2.6 compares categories of analysis techniques as described in
the two reviews and defines a high-level grouping for them.

<table>
<thead>
<tr>
<th>Group</th>
<th>Categories from [52]</th>
<th>Categories from [79]</th>
</tr>
</thead>
<tbody>
<tr>
<td>General analysis for intelligent tutoring systems</td>
<td>Model tracing of solution steps</td>
<td>Constraint-based modeling of a solution</td>
</tr>
<tr>
<td></td>
<td>Data analysis of large solution sets</td>
<td></td>
</tr>
<tr>
<td>Static program analysis</td>
<td>Basic static analysis of programs</td>
<td>Functionality testing of source code</td>
</tr>
<tr>
<td></td>
<td>Intention-based diagnosis</td>
<td>Code quality analysis</td>
</tr>
<tr>
<td></td>
<td>External tools</td>
<td>Software metrics</td>
</tr>
<tr>
<td></td>
<td>Other (AI methods)</td>
<td>Plagiarism checks</td>
</tr>
<tr>
<td>Transforming programs for further analysis</td>
<td>Program transformations</td>
<td></td>
</tr>
<tr>
<td>Dynamic program analysis</td>
<td>Automated program testing</td>
<td>Functionality testing of outcomes &amp; source code</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Test development</td>
</tr>
</tbody>
</table>
General analysis for intelligent tutoring systems have been applied to programming exercises. The review describes techniques that model program code or programming steps [18; 41; 44; 46; 51; 57]. In a technical sense, they use sophisticated static analysis.

Static program analysis includes remaining techniques that analyze program codes statically without executing them. The grouping of the review categories is based on the techniques they describe. Functionality testing of source code discusses some techniques that use dynamic analysis, and I decided to include it in both groups. Intention-based diagnosis attempts to model programmers' writing process more meticulously [47]. Essentially, these diagnoses recognize elements in the program code and use their model data to suggest next steps or other feedback.

Transforming programs for further analysis can help with many different techniques. Typically, transformation rules change program code without changing its function. Normalizing structures, such as expanding augmented assignments, is useful for simplifying higher-level analysis or relaxing program similarity checks.

Dynamic program analysis need to execute programs to analyze their dynamics. The most common automated assessment, as well as feedback generation technique, is developing automated tests for program requirements [52]. If multiple techniques are applied, tests are typically combined with basic static analysis [52]. This reflects how the industry is also complementing results of automated tests with knowledge from static analysis of program code.

2.5.1 Static Program Analysis

To analyze programs, the source code is parsed using the grammar defined for the inspected programming language. Syntax errors may prevent parsing the program properly and stop further analysis. A valid program is parsed into an abstract syntax tree (AST), which stores the program elements and their syntactic relations as a tree model. For example, Figure 2.1 presents a Python code snippet and an AST parsed from it.

Tree structures can be traversed using common traversal algorithms, which visit every node of the tree in some defined order. For example, depth-first search (DFS) goes as deep into the children as possible before visiting sibling nodes. While traversing each node in the tree, a stack structure can be used to retain context by storing references to every ancestor of the node that is currently being visited. Figure 2.1 visualizes
an example where an AST is being traversed using DFS and a stack.

One method to implement a program transformation is to use a recursive DFS function, which returns the nodes of the tree. Depending on the visited node and its context, the function can return the original node with its children transformed or a replacement node. Some AST libraries can export the modified tree model either directly as an executable object for the interpreter or as a program code that can be stored and executed. Other may require the developer to implement code writing from AST if needed.

The simplest form of program analysis is probably different software metrics, such as counts of certain types of elements in the AST. Those metrics aim to pack some characteristics of a program into single values without inspecting what the program does. Typically, metrics can be calculated efficiently and they could provide a rough estimate of similarity for programs that provide the same functionality [125]. For example, a student's program for an exercise could be measured to be significantly larger or more complicated than a certain model solution.

In education, current approaches to code quality often apply tools from industry that enforce style conventions and identify code patterns that are considered common mistakes, bugs, or security threats [38; 63]. Style conventions are stored as rules starting from the use of indentation, line breaks, and separators but can also concern the maximum size of functions.
or a comment block for each function signature. They can enforce syntax at the character level according to the elements in parsed AST.

In addition, code quality tools can search for unwanted patterns. For example, AST could be searched for loop nodes that have an assignment node as an ending condition. For several programming languages, those are potential incidents where assignment statements (\(=\)) are mistakenly used in place of an equality operator (\(==\)). Failing to escape user input in a database query is a typical security issue that could be detected via backtracking argument expressions for a database call in the AST. Some commercial tools apply AI, for example by training machine learning models, that can recognize potential code quality issues. Unwanted patterns are also referred to as code smells that can reveal code segments that may need refactoring to improve the design and maintainability of the program [80]. Because there is a level of uncertainty involved, quality tools often raise warnings aimed at professionals and offer means to suppress incidents.

Tools specifically created for programming education also inspect patterns in program code. Soloway and Ehrlich [114] define intention-based diagnosis as an attempt to model programmers’ plans that are implemented as desired structures in the AST. Similarly, constraint-based modeling can detect violations of expected constraints on AST and model tracing can suggest next steps when a state of the AST is recognized. It appears that typical modeling approaches are dependent on the exercise and on the possible program designs to complete it [41; 44; 47].

Static analysis can identify some general concepts that are above what is directly written in program code but are well below understanding of what the program does. For example, recursive functions or roles of variables can be identified in AST. Taherkhani et al. [125] analyzed the roles of variables and software metrics to detect implemented algorithms. Santos [100] analyzed the roles of variables to guide how program states are visualized more naturally.

Both data analysis and plagiarism checks use similarity measures of program codes. Syntactic similarity is analyzed using sequences of language components, which are referred to as tokens, in the source code [42; 122]. Structural similarity is measured using for example edit distance of ASTs, which counts the number of atomic edits needed to make the trees identical [76; 143]. Feedback for a student’s program is generated based on a similar model solution or a program state in a large data set of student
Background

codes for the same exercise. Once the current program state and the next desired state are determined, their differences can be processed into hints. Different AI tools could be trained with large data sets available on exercises but recent research on pre-trained models may offer a more generic approach. For example, Koutcheme et al. [53] employ automated program repair using open large language models (LLMs)\(^3\) to produce potential next steps to help students who are stuck in a programming exercise.

2.5.2 Dynamic Program Analysis

A prevalent strategy in automated assessment is to execute the program using either output comparison with expected output or different testing frameworks adopted from industry [52; 79]. Unit testing is used to check more granular assertions about separately executed program components, which potentially allows more detailed feedback than comparing output from the complete program. In web and mobile domains, frameworks that support automated testing via user interface have been employed [110; 14], which resembles system testing. In the education context, some tests may include random inputs to prevent students from hardcoding correct answers into their programs.

The tested component of the program is executed before any results can be inspected. If no runtime errors occurred, only results that fail a test remain to explain what happened. Backtracking the execution steps from the results to incorrect code may be an overwhelming task for a novice student having little knowledge of debugging. To alleviate this problem, Striewe and Goedicke [121] automate debug traces of variable values using the debugging interface of the runtime environment. This allows students to inspect values at each execution of a code line and identify where the values start to behave unexpectedly.

Designing test cases for programming exercises is a considerable effort but it may be at least partially automated. Li et al. [59] use dynamic symbolic execution\(^4\) to generate inputs for test cases that cover the majority of behaviors of a program, which in turn helps to quantify the behavioral similarity of two programs. This could be used as an alternative for

\(^3\)A probabilistic model, typically neural network, that is pre-trained to generate text for prompts and can power AI systems.

\(^4\)Symbolic execution uses a special execution engine to collect conditions for different execution paths. The results are necessarily imperfect as execution must be bound into a feasible number of paths. In addition, some calls, such as for the operating system, do not support symbolic execution. [33]
structural similarity of programs to generate feedback and hints.

Other dynamic analyses include measuring the performance of a program or quality of students’ test development. Performance in processor time or executed instructions may be used to check time complexity of a student’s program [135] or to order a set of programs by their performance. Few assessment tools can analyze a student’s test suite for their program using available code coverage tools [28; 29; 24]. Because code coverage does not necessarily measure the quality of developed tests, Aaltonen et al. [1] propose mutation analysis as an alternative. It transforms a program into several mutants by introducing errors and counts how many of them are caught in tests.

2.6 Automated Question Generation

Many current online courses depend on question pools to assess learning outcomes. The supply of fresh or personalized questions has motivated researchers to develop automation. Similarly, this dissertation proposes personalized questions, which depend on automated question generation. In their systematic literature review, Kurdi et al. [55] describe that questions are typically generated from a selected source that can be text or a more structured ontology, such as a medical database or linked data on the internet. The structured sources include relations and categories that support semantic approaches without the challenges of natural language processing. The authors of the review classify three distinct mechanisms of generation:

- **Templates** are question phrases that include placeholders, which are populated using facts extracted from an analysis of the source.

- **Rules** have been used to transform words of a selected sentence in the source to form a question. Words can be displayed as blanks to fill in or cleverly replaced with what, where, or who.

- **Statistical methods** use source as training data for machine learning models. After the review, this area has rapidly changed due to the success of pre-trained LLMs.

Kurdi et al. [55] list a few general challenges for the different mechanisms. Efforts for templates and rules focus on the design while training statistical methods require a curated data set. Templates offer limited linguistic
Background

diversity while rules or traditional statistical prediction produces sentences that share similarities with the source data.

LLMs can circumvent the traditional challenges. They are pre-trained to understand a variety of natural languages and common syntaxes, such as programming languages, in text format. LLMs do not need to be trained with the source data but they can read it as part of the input, analyze it as necessary, and generate output as an answer to a query. In that sense, they approach a definition of general-purpose AI. State-of-the-art LLMs have the capacity to generate fluent questions as answers to prompts that include large source data and a request to formulate questions about it.


*Selection* is responsible for choices before generating questions. Appropriate templates, rules, or prompts are selected based on the used source.

*Generation* applies the previous selection to the source using one of the generation mechanisms to produce questions. Depending on the mechanism the correct answer can be generated as well. For MCQs, generation includes also *distractors*, which are the incorrect answer options. Distractors are an active area of research [30; 34; 45; 105; 116]. Generally, the aim is to find distractors that are similar or somehow related to the correct answer without ignoring the fact they must be incorrect.

*Feedback* should be enabled to explain correct and incorrect responses. It is widely agreed that feedback has considerable potential to increase students’ learning by confirming and altering their knowledge [131]. Furthermore, providing explanations in addition to knowledge of the correctness or the correct answer is an effective strategy [131]. For MCQs, feedback can be generated in advance along with the answer options. Providing feedback for free text answers is considerably harder, although recent advances promise at least assistance to humans [101; 124]. However, open-ended questions can offer the freedom and time that are necessary to develop an understanding of abstract and advanced concepts. There seems to be a trade-off for the ability to provide automated feedback.

*Filtering* of imperfect questions based on their content may be required after they have been generated. The generated questions could be ranked and filtered by different criteria [55]. Linguistically questions should be at least fluent, unambiguous, and semantically correct. Educational considerations include difficulty, discrimination, cognitive level, relevance, and learning outcome.
Some examples of using automated question generation in the computer science domain exist. Shenoy et al. [108] generate variations of a seed problem, where a student is asked to construct deterministic finite automata, using mutation rules. Zavala and Mendoza [142] combine question templates with template-specific scripts, which generate both values for the placeholders in the template and the correct answer text. Thomas et al. [127] generate randomly expanded programs according to several parameters and students are asked to answer MCQ about the return value. They simulate program execution so that the correct answer is captured and distractors are generated from incorrect execution paths.
3. Questions About Learners’ Code (QLCs)

This chapter presents an approach to query students about their programs and discusses possible designs. The approach named questions about learners’ code is presented to the research community in PI, which also outlines a possible system to generate and deliver questions. Technical details are discussed at length and complemented with implemented open-source systems that have been introduced in PIII, PV, and PVI.

3.1 Definition

A lot of introductory programming courses employ a large number of small program writing exercises as discussed in Section 2.1. The practice of writing program code not only helps the learner build a routine but is also favored as a realistic activity by many students.

Considering the educator’s perspective, little is actually known about the students based on the practice programs they write. Students can apply their programming knowledge to systematically design and test a program. In contrast, many students start with code they search on the internet [140], copy from examples in the material, generate using AI tools, or get from a peer [3]. Students can fix possible errors in their code using repeated feedback from automated assessment [6] or direct advice from teaching assistants and peers without completely understanding their program. The students who understand how their program works have learned more than those who incautiously got a program to pass the required criteria, such as a set of functional tests.

Publication PI suggests that learning tasks can be developed to support the goal of students understanding and comprehending the program code they produce. Asking students questions about their program has the potential to assess the level of their current knowledge in programming. This
Questions About Learners’ Code (QLCs) dissertation presents how to devise many different automated questions by analyzing the structure and behavior of the program.

As an example, Figure 3.1 presents a set of questions that could be used to probe different levels of comprehension about a variable that a student has used in their program. The example questions range from recognizing elements and their relations in program syntax to describing the design and execution states of the program. Furthermore, they are all such that forming the questions as well as their correct answers can be automated as presented in this dissertation.

A targeted program

```python
1 def averageAllPositiveIntegers(lst):
2     count = 0
3     total = 0
4     for num in lst:
5         if num > 0:
6             count += 1
7             total += num
8     if count > 0:
9         return total / count
10    return 0
```

Example questions

- Which type of program element is `count`?
  - A variable
- On which line is variable `count` declared
  - Line 2
- What is the role of variable `count`?
  - A stepper (See Table 2.3)
- Which values and in which order are assigned to variable `count` when executing the function with argument `[2, -1, 4]`?
  - The values: 0, 1, 2

Figure 3.1. An example set of possible questions about a variable in the program.
Questions About Learners’ Code (QLCs)

An approach employing questions such as these is defined in PI as Questions about Learner’s Code (QLCs):

1. They are questions about program code that a student has written;
2. they refer to concrete constructs or patterns in the student’s program;
3. they are posed to the student themselves by a computer; and
4. they are automatically generated from an analysis of the student’s code.

The content of a question is up to the designer’s imagination, although practically technology limits the opportunities. Questions are based on knowledge that can be deduced automatically from the program code and possibly other supporting material, such as a model solution. More importantly, the design of QLCs is directed by the aim of prompting students to answer them. I discuss the aim and purpose of QLCs next.

3.2 Potential Benefits and Challenges

I have previously discussed the educators’ perspective to evaluate how well students understand the program code they created. This is one aim that can guide the design of the questions. As presented in Section 2.3, Block Model is an educationally oriented interpretation of program comprehension. It provides a framework where the QLCs can be designed to target different levels of abstraction and dimensions of understanding. To chart students’ range of understanding, the QLCs should target the full breadth and depth of Block Model. PI demonstrates using examples in Table 3.1 that questions can be devised for all dimensions at least up to the relational level. Based on careful design and available references [43], QLC can be expected to probe the targeted area of program comprehension. However, there may be unnoticed factors that affect the answers. Therefore, questions need to be evaluated.

A good long-term aim is to support students’ improvement in program comprehension. How does assessing program comprehension using QLCs help to reach that aim? First of all, assessment can have a direct influence on what students consider important or where they direct their efforts [99]. Therefore, a mere shift in what is assessed is meaningful for what students learn. In a way, QLCs make it students’ responsibility to display that they comprehend their own program. I see an analogy with how automated
Table 3.1. Example QLCs from PI that target different areas of program comprehension in Block Model.

<table>
<thead>
<tr>
<th>Area</th>
<th>Suggested QLC Template</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atom + Text</td>
<td>Which of the following are variable names in your function?</td>
<td>MCQ</td>
</tr>
<tr>
<td>Block + Text</td>
<td>A loop starts on line [N]. Enter the number of the last line inside this loop.</td>
<td>Value</td>
</tr>
<tr>
<td>Relational + Text</td>
<td>Line [N] uses a variable. Enter the line number where that variable is declared.</td>
<td>Value</td>
</tr>
<tr>
<td>Atom + Execution</td>
<td>What is assigned to variable [V] on line [N] when executing function [F] on expression [E]?</td>
<td>Value</td>
</tr>
<tr>
<td>Block + Execution</td>
<td>During the program execution, how many iterations are performed by the loop starting in line [N]?</td>
<td>Value</td>
</tr>
<tr>
<td>Relational + Execution</td>
<td>Which of the following best describes the role of your variable [V]?</td>
<td>MCQ</td>
</tr>
<tr>
<td>Relational + Execution</td>
<td>How deep does the call stack grow when executing function [F]?</td>
<td>Value</td>
</tr>
<tr>
<td>Atom + Function</td>
<td>Describe the purpose of the condition on line [N].</td>
<td>Open</td>
</tr>
<tr>
<td>Block + Function</td>
<td>Justify your choice of name [V] for the variable declared on line [N]—do you have a better suggestion?</td>
<td>Open</td>
</tr>
<tr>
<td>Block + Function</td>
<td>Select the part of your program that is responsible for [X]. <em>(The question could be generated from a subgoal annotated onto a model solution.)</em></td>
<td>Select</td>
</tr>
<tr>
<td>Relational + Function</td>
<td>Explain, in your own words, the purpose of the loop that begins on line [N], and how that loop helps method [M] accomplish its task.</td>
<td>Open</td>
</tr>
<tr>
<td>Relational + Function</td>
<td>Here is a little example program that has some similarities with yours. Select the part of your program that serves a similar purpose as the highlighted code in the example.</td>
<td>Select</td>
</tr>
</tbody>
</table>

assessment systems can require students to provide their own unit tests for their program, and then produce feedback on test coverage [29].

That being said, making program comprehension a learning objective and providing implicit instruction as well as versatile practice is the key. From an assessment perspective, successful QLC designs, which target specific concepts, could inform educators not only about students’ general

---

1The answer type: multiple choice (MCQ), a single value, or open-ended free text.
level but also about specific issues related to program comprehension. This allows educators to adjust the teaching and materials as required.

As an exercise, QLCs aim to guide students to actively analyze and understand the structure, logic, and purpose of the code they created. By posing questions about specific aspects of the code, such as variable usage, control flow, or execution states, QLCs encourage learners to engage in a deeper level of code reading. Therefore, I argue that QLCs have the potential to improve students’ program comprehension and the ability to effectively reason about and debug code. Considering students’ task types in Table 2.1, the QLCs we propose require declarative knowledge as well as comprehension and prediction skills. This does not indicate that it would be impossible to design QLCs that involve higher-level tasks, such as providing an example or model. From the perspective of the targeted programming task, QLCs have the potential to guide self-reflection.

As discussed in Section 2.4, students’ problem-solving process can be improved by making them aware of their progress [65] or enforcing completion of an important step [87]. Although previous think-aloud studies of programming tasks identified no metacognitive issues beyond program tests [86] and rarely deeper self-regulation behaviors [64], I argue that metacognitive support should extend to reflection. As discussed in Section 2.2, novices’ knowledge in programming is fragile. When novice creates a program, their comprehension of it may be elusive and hard to put into words. QLCs have the potential to support students in monitoring what they comprehend. QLCs potentially serve as prompts that may trigger self-explanation, for example, students may silently build deeper meanings and relations for newly acquired knowledge.

As discussed, there are potential benefits of applying QLCs but there are certainly also challenges. Only a handful of questions may be relevant to learners’ current objectives and prior knowledge. Pester students with tasks that are irrelevant to them at the given moment will likely annoy them and potentially hinder learning by diverting attention from what is actually important to the individual at a given time. Any new activity adds to the already significant workload for novice programmers or alternatively replaces other learning activities that may be more or as important for learning. Therefore, applying QLCs requires thorough design and consideration from the educators.

Acquiring evidence on many of the discussed potential benefits and challenges of QLCs would likely require longitudinal studies. Given the avail-
able resources, this dissertation needs to limit the scope to a few principal research questions as defined in Chapter 4.

3.3 Design Principles

This section lays out three principles for generating QLCs.

3.3.1 Unique Programs and Questions

According to the definition, QLCs are generated for each learner from their current program code. Therefore, each QLC is potentially unique and only asked once from one learner. This is radically different from conventional automated question generation. For example, questions generated for a pool, which can be used for selecting questions randomly, are typically valid and used for several cohorts of students. While conventional generation aims to produce large numbers of questions in advance [127; 142], QLCs must be generated separately for each learner and program they have created.

In practice, many programming assignments given to novice programmers are small, and programs they produce for a single assignment have limited variation, which enables feedback methods based on similar program codes [4; 76; 144]. To some degree, program elements could be predicted but the exact source code is never known beforehand. While it is possible to prepare generic questions for expected programs in advance as in PII, QLCs that refer to identifiers or line numbers in code must always be generated from the potentially unique program code. Treating programs as unique has an additional benefit in that the generation depends on what was programmed rather than what the learner was asked to program. Therefore, QLCs are by nature independent of the programming problem and can be applied to new exercises using the same generation system.

3.3.2 Error Free Questions

As discussed in Section 2.6, filtering questions by desired criteria is a sub-task in automated question generation. When managing traditional question pools, human judgment can typically be used to detect and filter unwanted questions from the pool. In that scenario, it may be acceptable that some poorly formulated or potentially incorrect claims appear in questions that are generated. As QLCs are unique and generated on
demand when learners complete their programs, educators cannot check
generated questions before they are posed to the learner. Because QLCs
make claims on programs where learners may have applied new concepts
for the first time, any false claims could confuse them badly. Therefore, the
questions should never have errors.

3.3.3 Automated Feedback

In the scope of the dissertation, QLCs are timely and integrated into
learners’ programming processes. QLCs aim to promote students’ metacog-
nitive awareness by prompting them to monitor their comprehension of
programming concepts they applied. Similarly, students have been pre-
viously supported by prompting them to monitor their problem-solving
stage [65]. I expect that immediate automated feedback on answers to
QLCs could support self-regulated learning. Feedback offers learners a
means to confirm their understanding before proceeding to new content.
Therefore, the ability to generate automated feedback should be considered
while designing QLCs. As discussed in Section 2.6, it is considerably easier
to support automated feedback for questions that have an exact answer
than for questions that expect open-ended answers. That makes MCQs
or questions expecting exact numbers or values attractive for developers.
As a downside, questions having exact answers are harder to develop for
higher levels of abstraction as suggested by the examples in Table 3.1.

3.4 System Design

I consider the requirement for the absence of errors a limiting factor for
QLC generation. Among the established question-generation mechanisms,
statistical methods have the least control over the produced questions.
Recent LLMs are also statistical and they are known to produce non-
factual output that contradicts the input. In fact, the phenomenon is called
hallucination and it is actively researched [73].

Nevertheless, a naive text prompt “Can you ask me a question about
the following program code?” would work as a starting point to gener-
ate QLCs using state-of-the-art LLMs. While this dissertation did not
experiment with this, I expect that control over generation improves with
new models and they may become viable, creative QLC generators in the
future. Although, sending students’ programs to corporations operating
state-of-the-art AI models is subject to data protection regulations as well as ethical discussion involving students.

Other mechanisms reviewed in Section 2.6 include rules and templates. I estimate transformation rules would essentially support QLCs that hide elements of the code and ask students to complete it. While this could be a practical test for comprehending simple concepts, I fear it is too limited for comprehension at depth. Furthermore, the student could well have a copy of the program, which they completed moments ago, readily available.

Question templates define questions as text, which in my experience can reflect any concepts that automated analysis is able to identify in the program. In contrast to statistical methods, a template defines exactly how QLC is formulated and the selection of templates determines what concepts can be asked about. Generation using templates is deterministic by nature. QLC developers can insert pseudorandom behavior by design and maintain a trustworthy expectation of how potential QLCs for some imaginary program would appear. Unfortunately, developing the necessary analysis for each template is a considerable effort, which limits the variation and types of questions that can be generated. Next, I present a template-based design we have applied in our technical contributions.

### 3.4.1 Template-Based QLC System

Figure 3.2 presents a design for a template-based QLC system as proposed in PI. QLC generation starts on demand once the student’s program code is received. In the scope of this dissertation, the program code has been assessed as functionally completed. For example, it has passed automated functional tests for the exercise.

The design separates program analysis into two paths, static and dynamic. Dynamic analysis requires some type of execution engine, while static analysis is more straightforward. The program code is parsed into an abstract syntax tree (AST), which can be searched for nodes of interest to produce static facts as discussed in Section 2.5. The execution engine is more complicated as the student code needs to be executed to collect dynamic facts during different states of execution. These facts may include the history of execution paths and values assigned to variables. If the programming problem expects user input or asks to declare a function, then a “test case” must be set up in order to execute and collect dynamic facts.

The generator has a collection of QLC types. Each QLC type is a combi-
Questions About Learners’ Code (QLCs)

Static Analyzer

Execution Engine

Test Case

Settings

Static Facts

Dynamic Facts

Student

Student’s Program Code

Set of QLCs

Question Generator

QLC Types

Question Templates

Figure 3.2. A simplified design for a template-based QLC system from PI.

nation of a question template, which is text including possible placeholders, and a program module to generate QLCs of that type. This dissertation refers to

QLC type as the design, which includes template and logic, and

QLC as the question instance, which targets a specific program.

First, the system determines possible QLCs, which are available for the targeted program using the necessary analysis. Some QLC types could offer multiple QLCs while some may not apply to the program at all and would then not offer any QLCs. For example, a QLC type could target loop structures and the program does not have any.

Next, the system selects individual QLCs to generate. Exercise-specific settings may disable some QLC types and limit the number of QLCs. Finally, the system generates the selected QLCs by populating templates using facts from the necessary analysis. In order to produce automated feedback, each QLC type should also generate a correct answer, distractors for MCQ, and explanations for all possible answers. Any of these could be text that includes placeholders to populate using more facts. The QLCs are then posed to the student to answer. Figure 3.2 simplifies the QLC system by ignoring the collection of students’ answers and delivery of feedback for them. Both are essential for a system implementation.
3.5 Question Design

The aims that guide the design of QLCs were discussed in section 3.2. The developed QLC types target different areas of program comprehension using Block Model [103] as a guide. The research has focused on novices, which draws attention to the lower levels of comprehension. In addition to the area of comprehension, other details, such as the complexity of the program code and the number of execution steps, affect the difficulty of QLCs. The internal logic of a QLC type could avoid elements that, for example, were assigned a value more times than a selected threshold. However, the person who selects the programming task and the QLC types to generate is ultimately deciding on the difficulty of the QLCs to ask.

Like any questions, QLCs should be fluent, unambiguous, and semantically correct. The included concepts and terminology should match teaching, learning materials, and announced learning outcomes. The formulation of a question template, that is, the question text, is important and likely not perfect on a first attempt. However, developing automated analysis, which produces facts for populating the template, is the greater effort in implementing a QLC type. The same analysis techniques that are used in automated feedback and were discussed in Section 2.5 can be used in QLCs. Next, I describe the program analysis we use and give a few examples.

3.5.1 Direct Facts in AST

The QLC systems in this dissertation are used for programs that have already been assessed as functionally correct. Consequently, analyzed programs should not have syntax errors and I only consider QLC generation for programs that can be parsed to ASTs.

Simple questions concerning syntax are generated on facts that can be directly searched from AST. Every parser, that we have used, produces AST nodes that store line numbers for their beginning and end positions in the source code. Following QLC template could apply on for and while nodes:

A program loop starts on line [N]. Which is the last line inside it?

The placeholder [N] in question and the correct answer is set to the line numbers for the selected node. Distractor lines are picked from other AST
nodes. Feedback for each option is generated based on the relative position of the node, for example, “This line is before the loop begins”.

3.5.2 Analysis of Naming Scopes

An additional step is required if naming, including variables or named functions, is involved. Programming languages have different rules for where a name binding is active. Typically, they have one or more types of program blocks that can declare local names that do not exist outside of their scope. This ensures that a local scope can safely bind a variable name without accidentally changing other variables that are already bound to the same name in some enclosing scopes. The analysis must respect the rules of the language and produce a list of functions and variables in the program, where each item should include name, scope information, declaration node, and reference nodes. AST tools, which provide this functionality, exist for a few programming languages, although I developed analysis for Python scopes. As an example, scope analysis allows to generate answer options for the QLC:

Which of the following are variable names in the program?

As distractors for this question, we have used built-in function names, which scope analysis can identify, reserved words, which belong to grammar for nodes used in the AST, and unused words, which are common variable names that were not used in the program at all. Feedback for each option is based on how it was generated using terminology and concepts that match the learning material.

3.5.3 Analysing Relations in AST

We develop some more advanced static analysis for recognizing structures that may raise and handle an exception or guard a block where zero would divide. As an advanced example, we consider analyzing the roles of variables, which were discussed in Section 2.3. Figure 3.3 presents a flowchart of possible decisions that an algorithm has to make to detect six different roles of variables. Several detection rules are loosely based on the descriptions by Santos [100].

More roles of variables have been defined [98] and the algorithm can only detect simple cases of the six roles. This is a decent solution for QLCs where it makes sense to ask about the ordinary cases in practice.
Figure 3.3. A flowchart describing an algorithm to detect six different roles of variables.

Programs. Detection of even a single variable role in the program enables the generation of a question in the form:

**Which of the following best describes the variable [X]?**

All answer options are one-sentence descriptions of variable roles. Depending on the selected variable, of which name replaces [X], one description is correct. As the short descriptions may not discriminate between the roles “Gatherer” and “Stepper”, the QLC type can prevent using one as a distractor for the other.
3.5.4 Collecting Data on Program Execution

In the realm of dynamic program analysis, we collected facts on how the program executes. We have experimented with two ways to achieve this.

First, program execution can be simulated, which gives full control over how the program executes and access to all related data. Consequently, the simulation is then responsible for the execution and must implement an interpreter for the programming language, which is a considerable effort and challenge. We took advantage of an existing library, Paddle, to model and simulate Java programs in PIII. This approach did simplify the generation of QLCs but limited programs to a subset of the programming language, which was supported in the simulation engine. A limited version of a programming language can support novice programmers learning goals, however.

Second, the program can be executed in the standard interpreter using instruments that collect data during the execution. Debugging environments, which are available for several programming languages, could support required data collection [121]. I did not experiment with this method, however. Program transformations offer another approach. We can transform the program so that its functionality does not change but calls are added to record data, which is in our interest. Table 3.2 presents transformations, which add instrumentation to record the history of variable values. Similarly, calls can be added to block structures, such as in looping or conditional statements, that keep a count of which paths are executed and how many times.

Table 3.2. An example of transforming AST nodes to record the history of each variable state. Instrument call signature is: rec(scope, id, value, [subsequent value])

<table>
<thead>
<tr>
<th>AST Node</th>
<th>Original</th>
<th>Transformed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
<td>\let a = &lt;expression&gt;</td>
<td>\let a = rec(0, 'a', &lt;expression&gt;)</td>
</tr>
<tr>
<td>Assignment</td>
<td>a = &lt;expression&gt;</td>
<td>a = rec(0, 'a', &lt;expression&gt;)</td>
</tr>
<tr>
<td>Compound assignment</td>
<td>a += &lt;expression&gt;</td>
<td>a = rec(0, 'a', a + &lt;expression&gt;)</td>
</tr>
<tr>
<td>Prefix update</td>
<td>++a</td>
<td>rec(0, 'a', ++a)</td>
</tr>
<tr>
<td>Update</td>
<td>a++</td>
<td>rec(0, 'a', a, ++a)</td>
</tr>
</tbody>
</table>

In addition, transformation must add an implementation of the instrument function, which in the example is named rec, to the executed code and make it available for any instrumented modules. The name binding of
the instrument must be selected so that it is not used in the original program. The responsibilities of the instrument function are to record the provided data for the supplied references and make it available for question generation at the end of executing the student’s program. The transformation approach has been used in **PV** and **PVI**.

Executing student code requires the same considerations as in automated assessment. Analyzed programs might have malevolent or accidental effects on the system. Therefore, they should be executed in a safe environment. When QLCs are implemented next to automated assessment or other feedback systems, we recommend taking advantage of the synergies and running question generation in a shared safe environment that produces QLCs, including questions, answer options, and feedback. Containerization is a current method that supports QLCs well [79]. Once dynamic facts have been collected, it is straightforward to generate QLCs using them:

Which values are assigned to the variable [X] when the program is executed?

The execution data contains the series of assignments to the variable selected as [X]. Distractors can be generated for example by manipulating some of the assignments.

### 3.5.5 Supporting Multiple Programming Languages

Programming languages have differences. Obviously, syntax may differ by keywords, operators, and separators. This alone could affect how questions should be formulated for a program in a given language. Configuring question templates by programming language would be similar to supporting multiple natural languages. Therefore, it would be straightforward to implement.

However, programming languages and their grammar can include concepts that are not available in some other languages. An example of that is how Python and Java implement *for* statements. Python always uses *for* to iterate values from an iterable structure, while in Java, *for* is similar to *while* statement, which manages variable values to step forward in iteration and to check the end condition. Pedagogical considerations apart, we could decide to avoid such problematic concepts in QLCs.

The differences would still exist in parsed AST, and analysis would be-
come increasingly complicated to support ASTs for different programming languages. Another option is to translate the targeted program to a supported programming language before parsing it. This is possible for novice programs that include a small subset of concepts. As an example, Python “for i in range(2)” is rather easy to map into Java. However, the equivalent “for i in [0, 1]” would better map to a different enhanced for structure, which iterates values in Java. Should we then teach two different for or do we duplicate all QLCs to both cases? Different exceptions to the rule start to build up quickly. We experimented with this and decided it is easier to handle separate generation systems for different programming languages.

3.6 Available Systems

This dissertation presents three QLC systems, which are outlined in Table 3.3. The main characteristics include the targeted programming language, the parser used to produce AST as described in Section 3.5.1, and the system used to simulate program execution or if the program is transformed before execution to collect data. Differences between the approaches, as well as Paddle, are discussed in Section 3.5.4. All of the systems are based on question templates as outlined in Section 3.4.1, and we have published their source code for open use in education and research.

Table 3.3. Available QLC generation systems.

<table>
<thead>
<tr>
<th>Name</th>
<th>Programming Language</th>
<th>Parser</th>
<th>Execution Engine</th>
<th>Related Publications</th>
</tr>
</thead>
<tbody>
<tr>
<td>JASK²</td>
<td>Java</td>
<td>ANTLR³</td>
<td>Paddle</td>
<td>PI, PIII</td>
</tr>
<tr>
<td>QLCJS⁴</td>
<td>JavaScript</td>
<td>Shift AST⁵</td>
<td>Transformed</td>
<td>PI, PV</td>
</tr>
<tr>
<td>QLCPY⁶</td>
<td>Python</td>
<td>ast module⁷</td>
<td>Transformed</td>
<td>PI, PVI</td>
</tr>
</tbody>
</table>

These systems only generate QLCs. However, the questions also need to be presented to students, students need means to answer, the answers need to be processed, and feedback needs to be returned to students.

Figure 3.4 presents a screen capture of a programming task where the

²https://github.com/tiagofmartinho/question-generator-engine
³https://www.antlr.org/
⁴https://github.com/teemulehtinen/qlcjs
⁵https://shift-ast.org/
⁶https://github.com/teemulehtinen/qlcpy
⁷https://docs.python.org/3/library/ast.html
Questions About Learners' Code (QLCs)

Figure 3.4. Using QLCJS in a programming task in PV. First, the student only saw the program editor (1). Once the student successfully completed the program, the source code was supplied to QLCJS. It generated three QLCs that were displayed for the student to answer (2).

A student has previously passed automated tests for the small program above and they are posed to answer three multiple-choice QLCs, which were generated using QLCJS. They answered the first QLC correctly and received feedback. Their first answer to the second QLC is incorrect and their feedback explains why. This dissertation does not discuss integration to different systems as technically that is system-specific. See PV for an example and discussion.
This chapter defines the research questions and scope of the dissertation. Additionally, it presents the overarching research design and how different studies contribute to the results.

4.1 Definitions

Chapter 3 presented a program comprehension exercise that targets students’ own programs retrospectively using small automated questions in a scalable way. Such new learning tasks might not work by design and could require different knowledge and efforts than estimated by the creator. Therefore, research is required to assess how students in a real context interact with a particular task and what it demands from the students to complete it. By answering the following research questions, this dissertation seeks foremost to help researchers in the development of QLCs and educators in the application of QLCs.

First of all, there are an unlimited number of designs for different kinds of QLCs and their difficulty likely differs. Students’ performance depends on how different QLCs are positioned over courses and should therefore be researched in different contexts. Optimally, QLCs are relevant, in contrast to trivial, for the learner. Furthermore, knowledge of the types of errors students frequently make helps to evaluate QLCs. In order to support students’ reflection, QLCs must be understood correctly, and their feedback should help to refute misconceptions rather than create new ones. The first research question focuses on students’ answers to QLCs. In addition, students’ perceptions about answering QLCs can inform about the effort and possible learning related to QLCs, although the possible biases need to be considered.
Research Questions

**RQ1** How do students answer different kinds of QLCs?

Knowing how students answer is different from knowing why they answer as they do. It is extremely hard, albeit necessary, to research what students are thinking. The second research question aims for insights into how answers to QLCs are actually related to programming skills and knowledge. The student’s overall success in the programming course is considered an indication of acquiring a level of programming knowledge. In addition to looking forward to predicting course success, this dissertation looks backward at how students create a program before answering QLCs about it.

**RQ2** To what extent can QLCs probe students’ knowledge and program comprehension?

Finally, recent advances in AI, or its availability to the public, are prevalent in current discussions about education both in media and among researchers. Students may be able to consult AI on simple program comprehension questions, but are there areas where novice programmers perform better? Furthermore, the current extent of program comprehension as displayed by AI can inform the use of AI for different programming-related tasks. The answer to the third research question helps to discuss how QLCs in the future might be used in conjunction with AI programming tools.

**RQ3** How AI answers different types of QLCs?

### 4.2 Scope

This dissertation is solely researching QLCs that target executable, functionally correct programs that have passed either automated assessment or have been solved in a guided lab session. In addition, we expect the student very recently created the program. In order to support QLCs at scale for recently completed programs, QLCs need to be produced automatically on demand. I believe applying QLCs to functionally incorrect or otherwise unfinished programs is a very interesting avenue of research. However, such designs would add more considerations to our research. For example, the elements that QLCs would target in an unfinished program should be
selected so that they are relevant to the sub-goals, which the programmer might have at the given time. Otherwise, students are likely to find the questions disturbing which would also affect their answers.

By targeting programs that have been recently assessed as correct this dissertation aims to test populations where students should have a decent, fresh, correct comprehension of their program. This reduces the amount of alternative explanations for incorrect answers to QLCs. Furthermore, the QLCs in the study aim to target students’ program comprehension rather than their related skills, such as synthesizing evaluations or other new artifacts based on their code. Considering our research questions, they ignore pedagogically interesting aspects. Most notably this dissertation does not evaluate how answering QLCs affects learning or self-regulation. Furthermore, this dissertation does not aim to validate any QLCs as instruments of specific program comprehension areas. We discuss this briefly in Section 8.6 about future work.

This restricted scope serves to deliver initial research findings on QLCs. That said, this work combines studies conducted with students in two universities in different countries and students participating in lifelong learning online. I consider this helps to generalize some of our findings.

### 4.3 Research Design

This section introduces the five empirical studies on QLCs. Table 4.1 presents a timeline of the studies, their learning contexts, the used programming languages, the answer method, and specific details of the research, which I will discuss in the section. While every study looks into success in answering QLCs, the variation of the learning context and overlaps in methodology work towards more trustworthy, generalizable answers to the research questions.

The studies PII, PIII, PV, and PVI quantitatively examine the numbers of different answers, and success rates at QLCs to answer RQ1. Because PII uses open-ended written answers, it includes qualitative coding of the answers to evaluate their correctness. Additionally, PIII quantitatively examines answers to a survey on student perceptions about answering QLCs.

Three studies, PII, PV, and PVI, research the correlation between successes in QLCs and other course tasks. PII and PVI perform statistical tests to validate differences between populations who answer QLCs cor-
Table 4.1. The timeline of QLC studies and their main characteristics.

<table>
<thead>
<tr>
<th>Year</th>
<th>Context(^1)</th>
<th>Language(^2)</th>
<th>Answer Type(^3)</th>
<th>Correlation(^4)</th>
<th>Additional Research</th>
</tr>
</thead>
<tbody>
<tr>
<td>PII</td>
<td>2021 Open</td>
<td>JS</td>
<td>Open</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>PIII</td>
<td>2022 CS1</td>
<td>Java</td>
<td>Value</td>
<td></td>
<td>Student perceptions</td>
</tr>
<tr>
<td>PV</td>
<td>2023 Open</td>
<td>JS</td>
<td>MCQ</td>
<td>Yes</td>
<td>Programming behaviors</td>
</tr>
<tr>
<td>PVI</td>
<td>2023 CS2</td>
<td>Python</td>
<td>MCQ</td>
<td>Yes</td>
<td>Program codes</td>
</tr>
<tr>
<td>PVII</td>
<td>2024</td>
<td>Python</td>
<td>MCQ</td>
<td></td>
<td>Artificial intelligence</td>
</tr>
</tbody>
</table>

rectly or incorrectly. PV compliments its statistical results with qualitative analysis, which observes the students’ processes of writing their programs before answering QLCs. These results are considered in answer to RQ2. PVI reflects on students’ errors in QLCs and the structure of their program code. We extend that discussion in Chapter 8.

PVII asks QLCs from AI about AI-generated programs to answer RQ3. The researchers assessed the answers’ correctness and qualitatively coded the types of errors in the answers. Similarly to RQ1, the numbers of different answers and success rates are examined.

---

\(^1\)The learning context: An open and online introductory programming course for lifelong learners (Open), an introductory programming course in a university in Portugal (CS1), and recap of prerequisite knowledge at second programming course in a university in Finland (CS2).

\(^2\)The programming language: JS is short for JavaScript.

\(^3\)The answer type: open-ended free text, a typed value, multiple choice (MCQ).

\(^4\)Examines correlation between successes in the QLCs and other course tasks.
5. Students’ Performance and Views on QLCs

This chapter examines students’ responses to QLCs by exploring various learning contexts and questions. To answer RQ1, the studies and empirical evidence from PII, PIII, PV, and PVI are presented.

5.1 From Theory to Practice

I have hypothesized how QLCs may be applied for programming education in Chapter 3 without testing them in practice. Empirical studies are needed to gather data on how students interact with different types of QLCs. Such a line of research can build insights into the demands and challenges that different types of QLCs place on students. Educators need to design QLCs that align with teachers’ learning objectives and that would optimally cater to different students’ individual learning needs. To get started with this challenging task pilot studies were conducted on three occasions while developing the technology for QLCs.

Questions that appear to be trivial for students, especially if repeated, are likely to burden the students and may steer their attention away from their current focus of learning. Questions that are more challenging than expected could be posed too early considering students’ current skills or they may include wording that is poorly designed. Looking at the answer content can help to differentiate these issues.

The contents of the collected answers to QLCs help to guide the early design, development, and research on the topic. Additionally, the answers could reveal cases where educators would like to offer additional support or improve specific parts of their teaching materials or design. However, more research and data are needed to ensure that QLCs in fact benefit students and support their programming education.

Next, I repeat RQ1 and define sub-questions that are answered by the
Students’ Performance and Views on QLCs

three pilot studies PII, PIII, PV and one later partial replication PVI.

<table>
<thead>
<tr>
<th>RQ1</th>
<th>How do students answer different kinds of QLCs?</th>
</tr>
</thead>
<tbody>
<tr>
<td>RQ1.1</td>
<td>How do students perform on different kinds of QLCs in different learning contexts?</td>
</tr>
<tr>
<td>RQ1.2</td>
<td>What types of errors are frequent in students’ answers to different kinds of QLCs in different learning contexts?</td>
</tr>
<tr>
<td>RQ1.3</td>
<td>How do students perceive the activity of answering QLCs?</td>
</tr>
</tbody>
</table>

5.2 Students’ Success on QLCs

This section presents the four studies researching students’ success in QLCs as well as the success rates they recorded. At the end, it briefly summarizes the result tables.

5.2.1 An Early Pilot Study

An early pilot study PII tested open-ended questions that were manually prepared to target programs that pass a selected programming exercise. The questions were designed to appear as QLCs but they were prepared before the course started based on the features that passing the functional tests for the related exercise required. The questions, listed in Table 5.1 along with targeted comprehension area and students’ success rates, were integrated into three programming exercises in JavaScript. They target parameters, return values, loop execution, event handlers, and program design. The study was conducted on an online introductory programming course for lifelong learners. Out of the students, who gave their research consent, 82 completed the first, 68 the second, and 33 the third exercise in this study. The instructor provided feedback based on the QLC answers to all students in two online sessions.

The study asks “How well do students answer QLCs?”, records answers for the eight QLC types, reports success rates for them as well as describes the extent of the answers, and quotes some common errors in them. The correctness of the open-text answers is assessed by the first author qualitatively. Inter-rater reliability is evaluated by the second author independently coding a fifth of the answers again and reaching substantial
Table 5.1. Manually prepared QLCs with the targeted program comprehension area in Block Model and students success rates in PII.

<table>
<thead>
<tr>
<th>ID</th>
<th>Area</th>
<th>Question</th>
<th>Success</th>
</tr>
</thead>
<tbody>
<tr>
<td>PII:1</td>
<td>Atom + Text</td>
<td>Which are the two function parameter names in your program?</td>
<td>95%</td>
</tr>
<tr>
<td>PII:2</td>
<td>Relational + Text</td>
<td>Where would you guess the parameter values come from when you test in the provided browser page synth.html?</td>
<td>67%</td>
</tr>
<tr>
<td>PII:3</td>
<td>Atom + Execution</td>
<td>How would you describe the difference between returning a value and printing a value using console.log?</td>
<td>72%</td>
</tr>
<tr>
<td>PII:4</td>
<td>Block + Execution</td>
<td>Describe the responsibilities of your outer loop in few words.</td>
<td>80%</td>
</tr>
<tr>
<td>PII:5</td>
<td>Block + Execution</td>
<td>Describe the responsibilities of your inner loop in few words.</td>
<td>75%</td>
</tr>
<tr>
<td>PII:6</td>
<td>Macro + Execution</td>
<td>Which parts of your program execute when browser opens the page?</td>
<td>58%</td>
</tr>
<tr>
<td>PII:7</td>
<td>Macro + Execution</td>
<td>Which parts of your program execute when the user clicks one of the note buttons?</td>
<td>79%</td>
</tr>
<tr>
<td>PII:8</td>
<td>Macro + Function</td>
<td>Describe in a few words how your program remembers the played notes.</td>
<td>91%</td>
</tr>
</tbody>
</table>

agreement (multirater $\kappa_{free}=0.86$, 95% CI [0.68-0.91]).

Additionally, the study asked “How does success in QLCs correlate with other learning data?” and compared student populations that answered QLCs correctly to those who did not. I discuss that in Chapter 6.

5.2.2 A Pilot Study With QLC Generation

A QLC tool for Java, named Jask, is presented in PIII along with a pilot study. Volunteers on a university-level introductory programming course used Jask on two occasions. The first had 102 and the second had 58 student participants. Students were instructed to supply the program they had created in a lab exercise to Jask and then answer QLCs that were generated for their program. Jask created each type of QLC, which was applicable to the targeted program, and students answered online by writing numbers, identifier names, and other values as requested in the question. Students received immediate feedback containing the correct responses.

Table 5.2 describes the different QLC types provided by Jask and the
success rates students achieved on the two rounds. The study divides Jask’s
QLC types into static and dynamic where the latter involves knowledge
of execution states of the program. The static questions PIII:1–12 target
call statements, parameters and variables, variable roles, and looping
structures. The dynamic questions PIII:13–17 target call stack depth,
function calls, variable assignments, and return values.

The study asks “How do students perform in QLCs on their lab exercises?”
and “How do students perceive the activity of answering QLCs?”. Its
results report success rates twice for every QLC type from two separate lab exercises. Additionally, the study queried students’ confidence for each answer and employed a small survey at the end of the course to research students’ perceptions of QLCs. I use the confidence and survey results to answer RQ1.3 in Section 5.4.

5.2.3 A Pilot Study With QLCs Integrated Into Exercises

In addition to presenting a QLC tool and exercises for JavaScript, PV includes a pilot study as well. The study integrates QLCs into four programming exercises on an introductory programming course for lifelong learners. Out of the students, who gave their research consent, 35 completed the first, 30 the second, 25 the third, and 18 the fourth exercise. In the studied exercises, students created their program in an online editor and tested it against automated functional tests. Once all tests were passed, the tool generated 2–3 QLCs, depending on the exercise.

Table 5.3 describes employed QLC types and students’ success rates. Questions target identifiers, passing values, looping structures, variable declaration, variable assignments, and object properties. Questions PV:4–6 are repeated twice over the exercises. The questions were displayed as MCQs below the program editor for the students to answer. Upon selecting an incorrect option, students received immediate feedback, which

<table>
<thead>
<tr>
<th>ID</th>
<th>QLC Template</th>
<th>Success</th>
</tr>
</thead>
<tbody>
<tr>
<td>PV:1</td>
<td>Which is the name of the function?</td>
<td>97%</td>
</tr>
<tr>
<td>PV:2</td>
<td>Which are the parameter names of the function?</td>
<td>83%</td>
</tr>
<tr>
<td>PV:3</td>
<td>Which value does [parameter] have when execution of [f(arg1, arg2, ...)] starts?</td>
<td>43% 76%</td>
</tr>
<tr>
<td>PV:4</td>
<td>A program loop starts on line [i]. Which is the last line inside it?</td>
<td>70% 56%</td>
</tr>
<tr>
<td>PV:5</td>
<td>A value is assigned to variable [v] on line [i]. On which line is [v] declared?</td>
<td>77% 88%</td>
</tr>
<tr>
<td>PV:6</td>
<td>Which is the ordered sequence of values that are assigned to variable [v] while executing [f(arg1, arg2, ...)]?</td>
<td>33% 56%</td>
</tr>
<tr>
<td>PV:7</td>
<td>Which best describes [method] on line [i]?</td>
<td>94%</td>
</tr>
</tbody>
</table>
explained their mistake. If the correct option was selected the feedback delivered explanations for each option. Figure 3.4 presents the students’ user interface. The students had an unlimited number of attempts to answer the QLCs.

The study considers student’s first choice among the answer options for comparing success rates with earlier studies where students could only answer once. Other results include the number of times that each different type of incorrect option was selected. In addition, the study analyzes the correlation between success in QLCs and success in the course. Furthermore, it extends the results with a qualitative analysis of the students’ recorded programming processes. I discuss the correlation and programming processes in Chapter 6.

5.2.4 Replicating on the Second Course

PVI applies QLCs in a new context. The study integrates multiple-choice QLCs into a recap exercise for Python at the start of a second university-level programming course. These students had completed their first programming course in Python. The study involved 291 students who gave their research consent and completed the exercise. The students had to first create a program for a variation of the classical Rainfall-problem [115]. They received automated feedback from functional tests to complete the program. Once the students had completed their program, they moved to the next task which prompted them to answer 2–3 QLCs. The number depended on which features their program included. Students could answer once and they received immediate feedback including explanations of each available answer option. Table 5.4 describes the three different QLC types in use, including the targeted comprehension area and students’ success

<table>
<thead>
<tr>
<th>ID</th>
<th>Area</th>
<th>QLC Template</th>
<th>Success</th>
</tr>
</thead>
<tbody>
<tr>
<td>PVI:1</td>
<td>Atom + Text</td>
<td>Which of the following are variable names in the program?</td>
<td>86%</td>
</tr>
<tr>
<td>PVI:2</td>
<td>Relational + Execution</td>
<td>From which line can program execution jump to line [i]?</td>
<td>85%</td>
</tr>
<tr>
<td>PVI:3</td>
<td>Atom + Function</td>
<td>Which of the following best describes the purpose of line [i]?</td>
<td>96%</td>
</tr>
</tbody>
</table>
rates. The study asks “How well do students answer QLCs for an exercise targeting prerequisite knowledge?” and reports success rates for each QLC type as well as the number of selected times for each incorrect option type. Therefore, it practically replicates the research question in **PII**, **PIII**, and **PV**, although using a different learning context—the second programming course instead of the first introductory programming course.

In addition, the study asks “How do the QLC results relate to success on a second programming course?” and compares the students who answered correctly to those who answered incorrectly for each QLC type separately. I discuss these results in Chapter 6.

### 5.2.5 Summary

All of the included studies on students’ answers to QLCs report the success rates at each different QLC type and each different programming exercise that the same QLC type may target. Individual success rates are presented in Tables 5.1, 5.2, 5.3, and 5.4. There is great variation in success rates inside the studies, that is, in each table. Furthermore, when the same QLC type is applied to two different exercises, the success rate can also considerably fall or grow as in PIII:10 or PV:3. Programs can include structures that add effort and complexity in comparison to answering an identically generated QLC in a simpler context.

In the studies combined, there are 35 evaluated QLC types in total. At the majority of them, that is 25 QLC types, students’ success rates are above 70%. Above 90% success rates are measured for 10 easier QLC types. With two exceptions, PII:8 and PVI:3, the easier QLCs are answered by recognizing single elements from the source code. There are also 9 difficult QLCs where students’ success rates fall below 70%. Besides PV:4 on one occasion, all the other 9 difficult QLCs require an understanding of program execution. The QLC types PIII:13–16 and PV:6 require multiple steps of tracing to deduce the answer and less than 50% of students answered correctly.
The results are summarized to answer RQ1.1.

**RQ1.1** How do students perform on different kinds of QLCs in different learning contexts?

The difficulty of a QLC type depends on the design but also on the targeted program. Typically, students have above 70% success rate at QLCs. However, less than half of the students successfully trace their own program code. The same results apply to students in lifelong learning and university courses. Students who have completed their first course do not have strikingly better success at comparable QLCs.

### 5.3 Frequent Errors in Answers to QLCs

The studies in PV and PVI employ MCQs and report the number of answers for each type of incorrect option. Those numbers are combined into Table 5.5. Next, I discuss these numbers by QLC type. I use the terms incorrect option and distractor interchangeably.

The QLC type PV:2 asks for the parameter names of a function. The name of the function and the “function” keyword are closely packed in a declaration with the parameter names. Those distractors were selected 13 times while variables were not selected once. Variables, which were declared in the function body, are deeper in the code structure than the parameter names. For PV:6, distractors that miss a single value of the correct sequence are selected 44 times. Unrelated sequences were only selected 8 times. The QLC types PV:5 and PVI:2 continue the pattern as the most frequent distractors are part of the targeted code structure. Out of the remaining QLC types three produced insignificant error counts. Distractors in PV:3, PV:4, and PVI:1 are selected evenly in this aspect.
### Table 5.5. The numbers of answers for each incorrect option in PV or PVI.

<table>
<thead>
<tr>
<th>ID</th>
<th>QLC Template and Incorrect Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>PV:1</td>
<td>Which is the name of the function?  &lt;br&gt;Keyword 1, Parameter name 0, Variable name 0</td>
</tr>
<tr>
<td>PV:2</td>
<td>Which are the parameter names of the function?  &lt;br&gt;Function name 7, Keyword 6, Variable name 0</td>
</tr>
<tr>
<td>PV:3</td>
<td>Which value does [parameter] have when execution of [f(arg1, arg2, ...)] starts?  &lt;br&gt;Random value 18, Parameter name 12, Literal in function body 8, Other parameter value 4</td>
</tr>
<tr>
<td>PV:4</td>
<td>A program loop starts on line [i]. Which is the last line inside it?  &lt;br&gt;Line inside (not last) 19, Before 13, After 3</td>
</tr>
<tr>
<td>PV:5</td>
<td>A value is assigned to variable [v] on line [i]. On which line is [v] declared?  &lt;br&gt;Other reference 20, No reference 4</td>
</tr>
<tr>
<td>PV:6</td>
<td>Which is the ordered sequence of values that are assigned to variable [v] while executing [f(arg1, arg2, ...)]?  &lt;br&gt;Miss last 21, Miss first 13, Random 8, Extra value 2</td>
</tr>
<tr>
<td>PV:7</td>
<td>Which best describes [method] on line [i]?  &lt;br&gt;Keyword 1, Operator 0, Argument 0, Parameter 0</td>
</tr>
<tr>
<td>PVI:1</td>
<td>Which of the following are variable names in the program?  &lt;br&gt;Miss a variable 22, Built-in function 21, Keyword 20, Unused word 3</td>
</tr>
<tr>
<td>PVI:2</td>
<td>From which line can program execution jump to line [i]?  &lt;br&gt;Try line 33, Outside try-block 3, Incorrect inside 0</td>
</tr>
<tr>
<td>PVI:3</td>
<td>Which of the following best describes the purpose of line [i]?  &lt;br&gt;Incorrect description 10</td>
</tr>
</tbody>
</table>

The results are summarized to answer RQ1.2.

**RQ1.2** What types of errors are frequent in students' answers to different kinds of QLCs in different learning contexts?

Typically, the frequently selected distractors are more closely connected to the structure and behavior the question targets than the rarely selected distractors.
5.4 Students’ Perception of QLCs

PIII presents two results to answer RQ3 about students’ perceptions. First, students could adjust their confidence level, from 1 to 5, before each answer they supplied to a QLC. Second, 44 students answered a survey at the end of the course. The survey questions and results are presented in Table 5.6.

Table 5.6. Results of a survey on students’ perceptions about answering QLCs in PIII, using a 5-point Likert scale from 1 representing low to 5 representing high. Additionally, students could select concepts for S3.

<table>
<thead>
<tr>
<th>ID</th>
<th>Question</th>
<th>Mean ± SD</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>When compared to other course tasks, how do you classify the necessary effort to answer QLCs?</td>
<td>2.7 ± 0.9</td>
</tr>
<tr>
<td>S2</td>
<td>When answering QLCs, how do you classify your comprehension of those questions (terms, etc)?</td>
<td>3.8 ± 0.9</td>
</tr>
<tr>
<td>S3</td>
<td>When answering QLCs, did you feel any learning or reinforcement of programming concepts?</td>
<td>3.5 ± 1.1</td>
</tr>
<tr>
<td></td>
<td>Which sort? Answers:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Terminology</td>
<td>89%</td>
</tr>
<tr>
<td></td>
<td>Loops</td>
<td>39%</td>
</tr>
<tr>
<td></td>
<td>Recursion</td>
<td>33%</td>
</tr>
<tr>
<td></td>
<td>Variable values</td>
<td>28%</td>
</tr>
<tr>
<td></td>
<td>Function definitions</td>
<td>17%</td>
</tr>
<tr>
<td></td>
<td>Function calls</td>
<td>17%</td>
</tr>
<tr>
<td>S4</td>
<td>To what extent do you find it useful to answer questions about your own code?</td>
<td>4.0 ± 0.9</td>
</tr>
</tbody>
</table>

Students answered four questions, S1–4, using a 5-point Likert scale. The abundant majority find the effort to answer QLCs similar or slightly less than for other course tasks (S1). The majority of students rate their comprehension of the QLCs at the middle or above (S2). Slightly more students rate their learning or reinforcement from QLCs in the middle but only few go below that (S3). When asked if QLCs were useful, students rate them higher (S4). Additionally, students could select programming concepts that in their experience QLCs helped to learn (S3). The majority selected terminology and approximately one-third selected also loops, recursion, and variable values.

Next, I summarize the confidence levels students reported for their QLC answers, which are presented separately for each QLC type in PIII. Excluding two question types, the mean of confidence ranges from 4.2 to 4.8 and has a standard deviation of roughly 1. The mean confidence drops
to 3.9 in question PIII:10 about fixed variables and to roughly 3.0 in PIII:13 about call stack depth. Both of these QLCs had also very low success rates and PIII notes there may have been a lack of instruction about call stack. The results are summarized to answer RQ1.3.

**RQ1.3** How do students perceive the activity of answering QLCs?

According to the survey answers, university students on an introductory programming course do not consider the effort to answer QLCs large in comparison with other course tasks. They think they comprehend the questions and are confident in answers even when success rates fall below 40%. They find QLCs useful and experience some learning, mostly regarding terminology.
6. Probing Students’ Knowledge With QLCs

After researching how students answer, this chapter is concerned with why they answer as they do. Similarly to Chapter 5, multiple studies in this dissertation PI, PIV, PV, and PVI contribute to the answer for the second research question.

6.1 From How to Why

Why do some students answer QLCs perfectly and others struggle to find the correct answer? The possible reasons for the poor performance range from lack of interest and ambiguous questions to not learning the concepts in the curriculum. There are multiple ways to approach the why question.

Often course success is used as a proxy measure of learning. However, measuring learning is rarely that straightforward. In fact, PI raises the point that students who pass programming exercises with perfect scores may understand their programs poorly. Regardless of the measuring issue, studies PI, PV, and PVI report a correlation between success on QLCs and success on the course or other assignments.

PIV is a working group report involving the steps that students take to solve a programming task. This work offers a view and discussion on different issues that students may run into when creating a program. On that background, PV qualitatively studies recordings of students’ stepwise programming processes. It identifies known issues in the processes and relates them with how students answered the QLCs about that program once it was finished.

Understanding why QLCs are answered incorrectly can validate their use and potentially help to identify students who need more help, and what kind of help needs to be provided. Next, I repeat RQ2 and formulate sub-questions based on the studies in this dissertation.
RQ2 To what extent can QLCs probe students’ knowledge and program comprehension?

RQ2.1 How does success in QLCs relate to course success?

RQ2.2 At which steps of writing a program do students experience challenges where experts would like to intervene?

RQ2.3 How do incorrect answers in QLCs relate to students’ steps of writing a program?

6.2 Relation With Success on Course

The studies in PII, PV, and PVI were described in Section 5.2. One part of each of those studies examines the relation between success in QLCs and success on the course, which this section describes in more detail.

PII studies three programming exercises, which were followed by QLCs. Figure 6.1 presents course success as three distributions vertically for each researched exercise (E1–E3). The first group of students (P1) did not successfully complete the programming task. The second (P2) completed the programming task but answered the related QLCs incorrectly and the third (P3) completed the programming task and answered QLCs correctly.

![Figure 6.1](image)

Figure 6.1. Course success after three programming exercises (E1–E3) from PII for students who failed (P1), passed only functional tests (P2), or passed both tests and QLCs (P3). Medians are marked in orange.
Each exercise has the students divided differently into groups (P1–P3).

The course success is defined as the ratio of exercise points collected from the four chapters that follow each researched exercise. The later chapters of the course introduce different concepts that are voluntary for passing the course and many lifelong learners only work on chapters of their interest regardless of their abilities. Therefore, success in exercises following the QLCs is a more trustworthy measure of their learning than success in all available exercises.

The exercises targeted the same students so groups over them are not independent. We do not analyze students who did not complete the programming task (P1). For each exercise separately, our null hypothesis is that median course successes in their P2 and P3 are equal. Course success is not normally distributed, and we use a nonparametric test. For the first exercise (E1), the median course success is significantly higher (Mann-Whitney U = 299, n₁ = 27, n₂ = 45, p < .001 two-tailed) for students who both created the program and answered the QLCs correctly (P3) than for those who failed at QLCs (P2). For E2 and E3, the median course success is statistically equal for students who answered QLCs correctly (P3) and for those who failed them (P2).

**PV** studied the same course as in **PII**, given again one year later. The QLC success was measured with a more granular scale using the average number of incorrect choices a student made while answering multiple-choice QLCs. The study found a moderate negative correlation (Pearson r = -0.459, p = 0.003) between average errors in QLCs and success on the course. The correlation is visualized in Figure 6.2. The course success was defined as the total points from the two first rounds of the course.

\[ \text{Figure 6.2. The distribution of students’ average number of errors while answering QLCs} \]

\[ \text{and correlation with the total points they received in PV. The regression line} \]

\[ \text{is marked in the middle diagram in orange. Quantiles } q_1 \text{ and } q_3 \text{ mark the} \]

\[ \text{limits for 25% of students with the least and most errors respectively.} \]
that formed the core of introductory programming. The following 4 rounds involved HTTP, sessions, asynchronous data, and using libraries. The 25% quantile having the most errors in QLCs hardly proceeds to the second round to continue programming studies.

**PVI** does similar statistical analysis as in **PII** on a second university-level course in programming. Figure 6.3 presents course point distributions for each of the three different QLC types (Q1–Q3) applied to one exercise.

![Histograms of course points](image)

**Figure 6.3.** Course point distributions for each QLC type (Q1–Q3) and whether students answered correctly or incorrectly from **PVI**. Tick marks denote grade limits in scale from 1 (pass) to 5 (excellent). Medians are marked in orange.

The first row represents students who answered the QLC correctly and the second row represents students who answered it incorrectly. The questions targeted the same students so groups are not independent over them, and each QLC type is analyzed individually. **PVI** does Mann-Whitney statistical U-tests for non-normal distributions using Alpha level $p < .05$ to reject the hypothesis of equal medians. Bonferroni correction for three separate tests $p/3 = .017$ is applied. The median course successes for the groups in Q1 are statistically identical ($U = 4932$, $n_T = 249$, $n_F = 42$, $p = .556$, CLES = .47). The median for those who answered the Q2 correctly is significantly higher ($U = 4716$, $n_T = 207$, $n_F = 36$, $p = .011$, CLES = .63) than for those who answered incorrectly. The same applies to Q3 with a large effect size ($U = 2124$, $n_T = 281$, $n_F = 10$, $p = .006$, CLES = .76).
The results are summarized to answer RQ2.1.

**RQ2.1** How does success in QLCs relate to course success?

The studies on an online course for lifelong learners suggest that students who answer repeatedly incorrectly to QLCs are in danger of dropping out of the course without learning introductory programming skills. In a university course, an early QLC about the execution order or purpose of a code line could identify students who are likely to receive lower course grades, even if they produced the same program as their peers.

### 6.3 Steps Students Take While Programming

A working group convened on a topic of steps that students take towards implementing a solution for a programming problem and wrote a report PIV, which is included in this dissertation. The aims of the working group were to:

- acquire datasets that store adequately granular steps on students writing programs;
- add expert annotation about when and how an instructor should intervene in the datasets; and
- compare the expert feedback to that given by learning environments.

This dissertation builds on the knowledge of the properties of suitable datasets for stepwise analysis as well as the knowledge of the different types and appearance of problematic steps. They guide the design of the study in PV. How to intervene at those steps and comparing the suggested expert feedback with that of the current learning environments is out of scope in this dissertation.

**PIV** identifies that stepwise analysis of program writing is supported in data where each addition or deletion of a token, which is close to a word in layman’s terms, is present or can be constructed from higher granularity edits, such as every keystroke. Individual logs, which include a single student’s work on a programming problem, comprise all program states from the start by every token edit until the student is finished with the task. The working group annotated 46 logs, that is, students’ program
writing processes, in two different datasets. Multiple experts annotated the researched logs. Annotations’ inter-rater reliability was calculated for multiple annotation rounds and substantial agreement was reached ($\kappa_1 = 0.563$, $\kappa_2 = 0.834$, $\kappa_3 = 0.71$, $\kappa_4 = 0.895$). Any conflicts were resolved in a discussion.

Based on the labeling experience and discussions, PIV develops guidelines for identifying when to intervene. The guidelines presented in Table 6.1 answer RQ2.2. PIV describes desired intervention points at the level that helps to assess or develop automated interventions. I inspect the guidelines so that a student is likely to experience challenges at and after the time when the ideal instructor would like to intervene. Then, semantic errors and deviation from specification are considered challenges at the time they appear as immediate intervention is recommended. When the program code is compiled or executed, any compiler or logical errors

<table>
<thead>
<tr>
<th>Event</th>
<th>Example</th>
<th>Intervention Point (when)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compiler error</td>
<td>Syntax or type error</td>
<td>If a student is not using the compiler often, or after the second compilation where the error goes unaddressed.</td>
</tr>
<tr>
<td>Semantic error</td>
<td>Syntax is correct but wrong semantics; e.g., $= \text{ instead of } \approx$</td>
<td>When a student moves to the next line, as these errors are hard to spot/debug.</td>
</tr>
<tr>
<td>Logical error</td>
<td></td>
<td>Once a student executes/tests the code or within 5 minutes if they choose not to run the code. If the error would lead to any further code being incorrect, intervene immediately.</td>
</tr>
<tr>
<td>Deviation from specification</td>
<td>Changing required function signature</td>
<td>When a student leaves the line.</td>
</tr>
<tr>
<td>Trial and Error behaviour</td>
<td>Iterating through conditional operands</td>
<td>Once it becomes clear the edits are guessing – not experimentation.</td>
</tr>
<tr>
<td>Hint or Feedback request</td>
<td>Pressing a “Hint” or “Show solution” button</td>
<td>Immediately when a student requests assistance.</td>
</tr>
<tr>
<td>Subgoal completion</td>
<td>Correct base case(s) for recursive function</td>
<td>When a student completes all steps of a subgoal.</td>
</tr>
</tbody>
</table>
should be considered challenges as intervention is required.

*Trial and error*, in other words tinkering, is the fuzzy area. Repeating edits could be useful experimentation until it is clear that guessing is involved. In that case, I consider tinkering harmful as it does not represent thoughtful action, which supports understanding and learning. If the progress seems to advance in coherent efforts to reach a *subgoal* at a time, I consider it a sign of control, which the guide recommends enforcing. *Hint or feedback* is not relevant for the retrospective inspection in **PV**.

While annotating the logs, experts recognized several programming processes that had plenty of repeating challenges. Such attempts can turn into “hundreds of steps that lead nowhere” as commented in **PIV**. Meanwhile, logs that did not involve challenges were often straightforward to the end. Perhaps the middle ground is most interesting where students depend on their ability to overcome the challenges they run into.

The results are summarized to answer RQ2.2.

RQ2.2 At which steps of writing a program do students experience challenges where experts would like to intervene?

Experts intervene immediately when a student deviates from the specification or has semantic errors. When the program is executed, any compiler or logical errors become challenges that may need intervention. Tinkering requires intervention when guessing becomes evident.

6.4 Relation With Students’ Programming Processes

**PV** conducts mixed methods research on students’ programming processes. First, the study quantitatively researched students’ success in both QLCs and the course as outlined in Section 5.2. Six subjects were selected for the qualitative part. They represent the full range of students in light of the quantitative results. As presented in Figure 6.4, four first students, S1–S4 had an increasing number of average errors per QLC and decreasing course success. S5 and S6 represent the outliers where success in the course does not follow success in QLCs. The grey marks denote students who copied-pasted the majority of their code to the online editor, thus preventing analysis of how it was created.

The researcher qualitatively analyzed how S1–S6 created their programs
and answered QLCs about them. Their programming processes were replayed in the editor as the instructor would look at the screen over the students’ shoulder while they worked. Every keypress is recreated from the logs with original delays. Based on the intervention guideline from Section 6.3, the researcher observed students’ work paying attention to their challenges. The results of the qualitative analysis are six accounts on S1–S6 working on the studied tasks. See PV for how each subject progressed in each of their programming tasks. This dissertation does not repeat the descriptions. Instead, I provide a brief summary of the results next.

No challenges were identified in the work of S1 and answers to QLCs were perfect. S2 was able to efficiently overcome typical challenges, both logical and compiler errors, and their first answer in QLCs that required tracing was a little off. S3 resorted to tinkering loop conditions for half an hour and was seemingly stuck with no progress in multiple exercises. In QLCs, S3 was incorrect about the variable’s declaration and tracing on two occasions. S4 started from a related example code and tinkered by replacing any numbers and operator signs to match the task description until they received help. S4 was mostly incorrect in QLCs. S5 ran into semantic and logical errors but could overcome these challenges. S5 answered QLCs promptly without much consideration and incorrectly a few times. S6 was challenged with syntax, in other words, compiler errors, which they eventually overcame in the first exercises before dropping out. S6 considered carefully and answered QLCs perfectly.
The results are summarized to answer RQ2.3.

**RQ2.3** How do incorrect answers in QLCs relate to students’ steps of writing a program?

Students who answered repeatedly incorrectly to QLCs run into a number of challenges in creating programs. The more incorrect answers in QLCs, the more illogical tinkering appeared. Those who could solve multiple challenges while programming did better in QLCs, although the care invested in answering has a significant effect on individual students’ success in QLCs.
The previous chapters focused on testing students’ performance and knowledge. This chapter momentarily turns away from students to research the extent to which state-of-the-art artificial intelligence (AI) understands program code. This chapter answers RQ3 based on PVII, which studies quantitatively correctness and qualitatively the types of mistakes in AI’s answers to QLCs.

7.1 To the Future

Recently, AI in the form of large language models (LLMs) has taken leaps forward. Since the summer of 2022, GitHub Copilot could be integrated into programming IDEs to suggest extensive amounts of program code from the mere intent of the programmer. Around the same time, tools that could generate images from short natural language prompts with artistic to photorealistic results started to become commonplace. As newspapers experimented with LLM-generated illustrations, arguments broke on fear of losing jobs. Discussions sparked about whether models had been trained with licensed materials.

By the end of 2022, ChatGPT—a conversational and easy interface to state-of-the-art AI—was launched. Approximately one month later it had 100 million users and the majority of Western audience knows what it is. Warnings appeared that white-collar jobs are at risk because this tool is so capable of automating writing tasks, and even that the development of AI should be paused to evaluate risks for humankind. Education is also changing as, in months, students and teachers became aware of an easy-to-use, free, and available tool to generate essays and complete other written tasks, including programming from natural language prompts.

Computing Education Research (CER) has reacted in a call to actively
shape challenges as well as opportunities for the era of AI-assisted programming [8], and in research efforts at conferences [70; 89].

A study on state-of-the-art LLMs answering different types of QLCs can shed light on multiple fronts. How readily students may consult AI on QLCs and is it rather harmful or beneficial for their learning? Where AI has weak areas if any in program comprehension? In what ways could QLCs and AI be used in education? How useful AI-generated datasets could be in CER? Based on the results we may discuss how QLCs and AI coexist in the future. Next, I repeat RQ3 and enumerate sub-questions from PVII, which helps to form a more comprehending answer for the primary question.

### RQ3
How AI answers different types of QLCs?

- **RQ3.1** How do large language models perform on program comprehension questions that have been generated from code created by large language models?

- **RQ3.2** What types of errors do large language models make when answering program comprehension questions?

### 7.2 Generation of Programs, Questions, and Answers

**PVII** hand picks six Python programming exercises from a publicly available exercise bank. A task description from each exercise is inserted into an LLM prompt asking to generate programs and then fed to the LLM. Ten unique generated programs are collected for each of the six exercises. In the next phase, the generated programs were read one by one into QLCPY, a QLC generation library introduced in Chapter 4. The library was used to generate every possible multiple-choice QLC of a unique type for each program. Altogether, 399 QLCs were acquired.

In the last generation phase, LLM prompts are constructed using a task description, a generated program code, and the first QLC about it. This prompt is fed into LLM to acquire an answer for the QLC. Keeping this chat history, QLCs are appended and sent one by one until all QLCs for the program are answered. The study programmatically queried answers from both the model currently employed in freely available ChatGPT known as “gpt-3.5-turbo” and “gpt-4”, which is currently available in OpenAI’s billed
services. The next sections discuss how this data including 798 answers from state-of-the-art LLMs is analysed.

### 7.3 AIs' Success on QLCs

Two researchers assessed the answers’ correctness by hand. The answer was deemed correct if it clearly indicated the correct option by either letter or label, without suggesting any incorrect options. Table 7.1 reports success rates of both LLMs for each QLC type. Both models answer QLCs fairly well. At worst, GPT-3.5 has below 70% success rates for QLC types PVII:3, PVII:4, PVII:7, and PVII:8. GPT-4 greatly reduces errors on all QLC types, although success rates for the same types where GPT-3.5 suffered remain a little below average for GPT-4 as well.

Table 7.1. Generated QLC types from PVII with the targeted program comprehension area in Block Model, the number of generations (N), and success rates for the two LLMs.

<table>
<thead>
<tr>
<th>ID</th>
<th>Area</th>
<th>QLC Template</th>
<th>N</th>
<th>GPT-3.5</th>
<th>GPT-4</th>
</tr>
</thead>
<tbody>
<tr>
<td>PVII:1</td>
<td>Atom + Text</td>
<td>Which of the following are parameter names of the function declared on line [i]?</td>
<td>60</td>
<td>95%</td>
<td>100%</td>
</tr>
<tr>
<td>PVII:2</td>
<td>Atom + Text</td>
<td>Which of the following are variable names in the program?</td>
<td>60</td>
<td>73%</td>
<td>98%</td>
</tr>
<tr>
<td>PVII:3</td>
<td>Block + Text</td>
<td>A program loop starts on line [i]. Which is the last line inside it?</td>
<td>50</td>
<td>40%</td>
<td>72%</td>
</tr>
<tr>
<td>PVII:4</td>
<td>Relational + Text</td>
<td>A value is assigned to variable [v] on line [i]. On which line is [v] created?</td>
<td>60</td>
<td>63%</td>
<td>93%</td>
</tr>
<tr>
<td>PVII:5</td>
<td>Relational + Execution</td>
<td>Which of the following best describes the role of variable [v] that is created on line [i]?</td>
<td>47</td>
<td>83%</td>
<td>91%</td>
</tr>
<tr>
<td>PVII:6</td>
<td>Atom + Function</td>
<td>Which of the following best describes the purpose of line [i]?</td>
<td>16</td>
<td>88%</td>
<td>100%</td>
</tr>
<tr>
<td>PVII:7</td>
<td>Block + Execution</td>
<td>Line [i] has a loop structure. How many times does the loop execute when running [f(arg1, arg2, ...)]?</td>
<td>50</td>
<td>68%</td>
<td>86%</td>
</tr>
<tr>
<td>PVII:8</td>
<td>Atom + Execution</td>
<td>Line [i] declares a variable named [v]. Which values and in which order are assigned to the variable when running [f(arg1, arg2, ...)]?</td>
<td>56</td>
<td>55%</td>
<td>70%</td>
</tr>
<tr>
<td><strong>Over all types</strong></td>
<td></td>
<td></td>
<td><strong>399</strong></td>
<td><strong>69%</strong></td>
<td><strong>88%</strong></td>
</tr>
</tbody>
</table>
Figure 7.1. Success rates by QLC type and programming task for the two LLMs from PVII. Darker red marks lower success.

Figure 7.1 presents success rates by both QLC type and programming task. Overall, the success rates decrease a little when the task towards T6 generates more complexity for the programs. Perhaps surprisingly, there are two combinations QLC type 8 for T1 and QLC type 3 for T6 where GPT-4’s success suddenly falls to 30% or below. GPT-3.5 performs worst at other combinations.

The results are summarized to answer RQ3.1.

RQ3.1 How do large language models perform on program comprehension questions that have been generated from code created by large language models?

Although the questions targeted code written by an LLM, both LLMs had errors in their answers. Overall, GPT-3.5 performs decently and GPT-4 improves greatly. Yet, at specific tasks and QLC types, the performance plummeted and this was model-specific.

7.4 AIs’ Errors on QLCs

After marking the correctness, two researchers met to discuss the observed errors and to develop a coding manual for them. Then, the two researchers annotated one-third of the data. They had a moderate inter-rater agreement (Cohen’s $\kappa = 0.501$) but half of the disagreements were dependent on the interpretation of a single code definition. The researchers resolved disagreements in a discussion and adjusted the coding manual which is
Table 7.2. The coding developed in PVII for incorrect answers from the LLMs to the QLCs.

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>a. Illogical execution step(s) described</td>
<td>Model describes the program's execution so that it differs from how the program actually executes.</td>
</tr>
<tr>
<td>b. Line number counted incorrectly</td>
<td>Model describes line contents that do not match with the line number that the model or the QLC referred to.</td>
</tr>
<tr>
<td>c. Interpreted question differently</td>
<td>Model answers logically but to a somewhat different question than posed.</td>
</tr>
<tr>
<td>d. Incorrect answer after valid explanation</td>
<td>Model explains correctly and sufficiently but finally selects a contradicting answer for an unknown reason.</td>
</tr>
<tr>
<td>e. Insufficient level of analysis</td>
<td>Model does not explain execution or context to the level of detail required to answer.</td>
</tr>
<tr>
<td>f. Valid explanation after incorrect answer</td>
<td>Model generates incorrect answer first but continues with contradicting, correct, and sufficient explanation.</td>
</tr>
<tr>
<td>g. No explanation available</td>
<td>Model did not generate any explanation other than the incorrect answer option(s).</td>
</tr>
<tr>
<td>h. Misconception about code element</td>
<td>Model describes a named code element as something else than it actually is.</td>
</tr>
<tr>
<td>i. The answer is not among the options</td>
<td>Model generates an answer option that was not offered in the multiple-choice question.</td>
</tr>
<tr>
<td>j. Hallucinates to justify incorrect answer</td>
<td>Model generates incorrect answer first and continues with a correct explanation until it suddenly starts to hallucinate to justify its answer.</td>
</tr>
</tbody>
</table>

presented in Table 7.2. Finally, the rest of the errors were coded.

Table 7.3 reports the number of incorrect answers for both models by error code. The majority of GPT-3.5 errors were either describing illegal execution steps (code a) or counting line numbers incorrectly (code b). GPT-4 mitigates the errors considerably so that 4 error codes disappear completely. However, GPT-4 frequently hallucinates explanations, which are unfaithful to the input, to stand by the selected answer.

Figure 7.2 presents numbers of incorrect answers by both QLC type and error code. Illogical execution steps (code a) are as common for QLC type 3 about the end of a loop as they are for types 7 and 8, which actually target execution. Unsurprisingly, the QLC types 3 and 4, which ask for line
Table 7.3. Number of incorrect answers from the two LLMs for each error code from PVII.

<table>
<thead>
<tr>
<th>Error Code</th>
<th>GPT-3.5</th>
<th>GPT-4</th>
</tr>
</thead>
<tbody>
<tr>
<td>a. Illogical execution step(s) described</td>
<td>31</td>
<td>10</td>
</tr>
<tr>
<td>b. Line number counted incorrectly</td>
<td>30</td>
<td>13</td>
</tr>
<tr>
<td>c. Interpreted question differently</td>
<td>12</td>
<td>6</td>
</tr>
<tr>
<td>d. Incorrect answer after valid explanation</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>e. Insufficient level of analysis</td>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>f. Valid explanation after incorrect answer</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>g. No explanation available</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>h. Misconception about code element</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>i. Answer is not among the options</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>j. Hallucinates to justify incorrect answer</td>
<td>3</td>
<td>14</td>
</tr>
</tbody>
</table>

Figure 7.2. Number of incorrect answers by QLC type and error code for the two LLMs from PVII. Darker red marks the higher frequency of the error code.

numbers, have the most errors about them (b). Both models can interpret differently (c) what the QLCs mean by “executing a loop” in type 7 or “variable role” in type 5. GPT-3.5 can easily handle QLCs about variables at insufficient level (e). It happens albeit rarely to GPT-4 on type 8 about variable tracing.
The results are summarized to answer RQ3.2.

**RQ3.2** What types of errors do large language models make when answering program comprehension questions?

Typical errors that both LLMs have are surprisingly similar to human errors, including illogical execution steps, interpreting questions differently, and conducting insufficient levels of analysis. In addition, both models struggle with line numbers in a way that humans are not likely to do. GPT-4 follows instruction much more strictly than GPT-3.5, which as a side-effect of our prompting causes GPT-4 to offer made-up explanations to validate itself. GPT-3.5 has more variate styles of explaining or not explaining answers.
8. Discussion

This chapter discusses the work in this dissertation, including its limitations and future implications. It discusses the results from Chapters 5 to 7 and examines them in the frame of the related research, which was summarized in Chapter 2.

8.1 The Evaluated QLC Types

Section 5.2 presented four studies, PII, PIII, PV, and PVI, on students answering QLCs. The QLC types between these studies have variations in design and wording due to different programming languages and generation systems. For the ease of the reader, Table 8.1 repeats all the QLC types, which have been evaluated in the dissertation, and already presented as four separate tables in Section 5.2. To generalize the analysis, I classify the questions and then synthesize the discussion over the results of the four papers. Previously, Bloom’s taxonomy has been used to both develop and classify programming questions, but some difficulties have been reported in its use [72]. In addition, the related work in Section 2.3 has used SOLO taxonomy [138] and neo-Piagetian development stages [126] to analyze cognitive abilities from students answers. I presented Block Model [103] in Section 2.3. Izu et al. [43] have used Block Model to classify program comprehension tasks and provide several example tasks for different cells in the model. PI develops example QLC questions that cover a large area of Block Model. I use this model to describe the evaluated QLC types and their relations.

PII and PVI already locate their QLC types in Block Model. In Table 8.2, I follow their example, as well as discussion in PI and the related work, to locate the QLC types in PIII and PV based on their contents. The evaluated QLCs start from the cognitively least demanding levels of abstraction.
Table 8.1. All QLC types, which have been evaluated in the dissertation, and students’ success rates.

<table>
<thead>
<tr>
<th>ID</th>
<th>Question</th>
<th>Success</th>
</tr>
</thead>
<tbody>
<tr>
<td>PII:1</td>
<td>Which are the two function parameter names in your program?</td>
<td>95%</td>
</tr>
<tr>
<td>PII:2</td>
<td>Where would you guess the parameter values come from when you test in the provided browser page synth.html?</td>
<td>67%</td>
</tr>
<tr>
<td>PII:3</td>
<td>How would you describe the difference between returning a value and printing a value using console.log?</td>
<td>72%</td>
</tr>
<tr>
<td>PII:4</td>
<td>Describe the responsibilities of your outer loop in few words.</td>
<td>80%</td>
</tr>
<tr>
<td>PII:5</td>
<td>Describe the responsibilities of your inner loop in few words.</td>
<td>75%</td>
</tr>
<tr>
<td>PII:6</td>
<td>Which parts of your program execute when browser opens the page?</td>
<td>58%</td>
</tr>
<tr>
<td>PII:7</td>
<td>Which parts of your program execute when the user clicks one of the note buttons?</td>
<td>79%</td>
</tr>
<tr>
<td>PII:8</td>
<td>Describe in a few words how your program remembers the played notes.</td>
<td>91%</td>
</tr>
<tr>
<td>PIII:1</td>
<td>Does function [f] depend on other functions?</td>
<td>98%</td>
</tr>
<tr>
<td>PIII:2</td>
<td>How many functions does function [f] depend on?</td>
<td>96%</td>
</tr>
<tr>
<td>PIII:3</td>
<td>Which other functions does function [f] depend on?</td>
<td>88%</td>
</tr>
<tr>
<td>PIII:4</td>
<td>Is function [f] recursive?</td>
<td>90%</td>
</tr>
<tr>
<td>PIII:5</td>
<td>How many parameters does function [f] have?</td>
<td>93%</td>
</tr>
<tr>
<td>PIII:6</td>
<td>Which are the parameter names of function [f]?</td>
<td>88%</td>
</tr>
<tr>
<td>PIII:7</td>
<td>How many variables (not including parameters) does function [f] have?</td>
<td>96%</td>
</tr>
<tr>
<td>PIII:8</td>
<td>Which are the variable names (not including parameters) of function [f]?</td>
<td>79%</td>
</tr>
<tr>
<td>PIII:9</td>
<td>Which variable will hold the return value of function [f]?</td>
<td>89%</td>
</tr>
<tr>
<td>PIII:10</td>
<td>Which are the fixed value variables of function [f]?</td>
<td>59%</td>
</tr>
<tr>
<td>PIII:11</td>
<td>What is the role of variable [v] in function [f]?</td>
<td>79%</td>
</tr>
<tr>
<td>PIII:12</td>
<td>How many loops does function [f] have?</td>
<td>86%</td>
</tr>
<tr>
<td>PIII:13</td>
<td>What was the maximum call stack depth when calling ([f(arg1, arg2, ...)])?</td>
<td>17%</td>
</tr>
<tr>
<td>PIII:14</td>
<td>How many function executions are performed when calling ([f(arg1, arg2, ...)])?</td>
<td>20%</td>
</tr>
<tr>
<td>PIII:15</td>
<td>How many times is variable [v] assigned when calling ([f(arg1, arg2, ...)])?</td>
<td>40%</td>
</tr>
<tr>
<td>PIII:16</td>
<td>Which is the sequence of values taken by variable [v] when calling ([f(arg1, arg2, ...)])?</td>
<td>29%</td>
</tr>
<tr>
<td>PIII:17</td>
<td>What is the value returned by the function call ([f(arg1, arg2, ...)])?</td>
<td>89%</td>
</tr>
<tr>
<td>PV:1</td>
<td>Which is the name of the function?</td>
<td>97%</td>
</tr>
<tr>
<td>PV:2</td>
<td>Which are the parameter names of the function?</td>
<td>83%</td>
</tr>
<tr>
<td>PV:3</td>
<td>Which value does [parameter] have when execution of ([f(arg1, arg2, ...)]) starts?</td>
<td>43%</td>
</tr>
<tr>
<td>PV:4</td>
<td>A program loop starts on line [i]. Which is the last line inside it?</td>
<td>70%</td>
</tr>
<tr>
<td>PV:5</td>
<td>A value is assigned to variable [v] on line [i]. On which line is [v] declared?</td>
<td>77%</td>
</tr>
<tr>
<td>PV:6</td>
<td>Which is the ordered sequence of values that are assigned to variable [v] while executing ([f(arg1, arg2, ...)])?</td>
<td>33%</td>
</tr>
<tr>
<td>PV:7</td>
<td>Which best describes [method] on line [i]?</td>
<td>94%</td>
</tr>
<tr>
<td>PVI:1</td>
<td>Which of the following are variable names in the program?</td>
<td>86%</td>
</tr>
<tr>
<td>PVI:2</td>
<td>From which line can program execution jump to line [i]?</td>
<td>85%</td>
</tr>
<tr>
<td>PVI:3</td>
<td>Which of the following best describes the purpose of line [i]?</td>
<td>96%</td>
</tr>
</tbody>
</table>
Table 8.2. Evaluated QLC types located to Block Model.

<table>
<thead>
<tr>
<th>Macro</th>
<th>Relational</th>
<th>Block</th>
<th>Atom</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PII:2</td>
<td>PIII:12</td>
<td>PII:1</td>
</tr>
<tr>
<td></td>
<td>PII:6</td>
<td>PIII:13</td>
<td>PIII:1</td>
</tr>
<tr>
<td></td>
<td>PII:7</td>
<td>PIII:16</td>
<td>PIII:2</td>
</tr>
<tr>
<td></td>
<td>PII:8</td>
<td>PIII:17</td>
<td>PV:1</td>
</tr>
<tr>
<td></td>
<td>PV:5</td>
<td>PIII:18</td>
<td>PV:2</td>
</tr>
<tr>
<td></td>
<td>PV:1</td>
<td>PIII:20</td>
<td>PVI:1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Macro</th>
<th>Execution</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atom</td>
<td>PII:3</td>
<td>PVI:3</td>
</tr>
<tr>
<td>Relational</td>
<td>PIII:4</td>
<td>PV:4</td>
</tr>
<tr>
<td>Block</td>
<td>PII:4</td>
<td>PIII:5</td>
</tr>
<tr>
<td>Atom</td>
<td>PII:5</td>
<td>PIII:6</td>
</tr>
</tbody>
</table>

Figure 8.1. Students’ success rates for each QLC type and researched programming task, arranged vertically by the targeted program comprehension area. Where visible, numbers in the colored marks refer to the QLC types presented in this dissertation.
Discussion

at the bottom left, where questions require structural understanding (Text) of single elements (Atom) at the time. Different studies have developed QLCs that reach up towards addressing the program holistically (Macro) and right towards understanding the purpose of the targeted sections (Function). QLCs for those extremes at the top and far right are somewhat harder to automate, at least by MCQs or exact answers, which explains the low number of QLCs there.

Considering neo-Piagetian stages or the related SOLO levels as applied in Table 2.5, cognitive demands in Block Model reach early concrete form when moving out from the bottom left corner. Concrete operational form or the SOLO category “relational” is reached around “Relational + Function”. I quoted Lister [60] in Section 2.3 about difficulties in inferring formal operational reasoning using the produced code as the only evidence. We did not attempt this but it may be possible to design QLCs that target the detection of formal operational reasoning. However, the questions may need to be open-ended or include additional exercise-specific material to ask about the program in a new context.

8.2 Tracing Is Hard, Even for One’s Own Program

Figure 8.1 presents the students’ success rates for each QLC type as circular marks colored by the study and numbered as in Tables 8.1 and 8.2. Horizontally the circles mark the success rate in the studied population and vertically they are arranged by the related cell in Block Model. For the majority of the QLC types, 70–90% of students answered correctly. When the comprehension area moves towards increased abstraction, such as “Block” and “Relational”, or towards higher dimensions, such as understanding execution, the mean of the success rate decreases. However, only one feature has notable evidence in the data. For a group of QLC types in the “Relational + Execution” area, only 20–40% of the students answered correctly. These are the QLCs that required multiple steps of tracing to deduce the correct answer (RQ1.1) as observed in Section 5.2.

Considering related work, the success rates are not surprising. Tracing is known as a demanding task for novices [82; 61; 67; 133]. According to our results, students are not significantly better at tracing program code they created themselves. Previous studies show that students have fragile programming knowledge through their first semester of programming [69; 74; 75; 129]. Further research points to that students need
Discussion

several weeks to reach the concrete operational form of reasoning in neo-Piagetian terms [19; 109; 126; 138]. Before that, they will have issues with program structures and the ability to abstract beyond one point of code. In our data, roughly 20% of students answer incorrectly to simple questions with such requirements.

Questions in “Atom + Text” primarily require terminological knowledge. Moreover, according to a survey in Section 5.4 students commonly had a perception of learning terminology while answering QLCs (RQ1.3), which suggests that gaps in terminological knowledge can explain some of the incorrect answers. Knowing the correct terms for each element is not a requirement to reason how the program works. However, I argue that some questions at “Relational” level, PIII:9, PV:5, or at “Execution” dimension, PII:4, PVI:2, ask for concepts that are critical for reasoning about the program. I expected that students, who successfully created a program, should answer them correctly. Yet 10–20% of those students fail. Overestimating students’ abilities and underestimating cognitive demands for programming exercises have a long history [75; 138], which suggests that teaching may focus on program writing at cost of other critical skills [141].

Success rates have significant variation and I must consider how trustworthy the results are. According to a student survey in PIII, the effort to answer is perceived acceptable and questions were comprehensible (RQ1.3). In Section 5.3, analysis of incorrect multiple choice answers, over two studies PV and PVI, suggest that incorrect answers are not randomly distributed but lean towards the correct answer (RQ1.2). Furthermore, analysis of individual students in PV supports that typical students spent time considering their answers. In this light, the design of QLCs, including possible distractors, and the nature of the targeted program are major factors for the success rates in our study.
The results are synthesized to answer RQ1.

**RQ1** How do students answer different kinds of QLCs?

While some incorrect answers to QLCs can be explained by poor knowledge of terminology, up to 20% of novice students answer incorrectly to questions, which I consider critical for reasoning how the program, which the students were able to correctly create, works. Such students may still develop their cognitive ability to reason about programs in concrete operational form. In addition, more than half of all students fail to trace their own program code, which resembles previous results in tracing example code.

### 8.3 Early Warnings About Students’ Fragile Knowledge

In a traditional programming exercise, students write a program according to the provided specifications and are assessed for the created program. Appending QLCs to the exercise yields additional knowledge of students’ abilities. Some of the students experience unproductive success where they manage to create an acceptable program but do not learn the concepts that the exercise was designed to teach [49]. The results in PII, PV, and PVI indicate that students, who create a program and answer QLCs about it correctly, differ from students, who create a program and answer their QLCs incorrectly.

While single QLCs may not produce significant correlations, lifelong learners, who repeatedly answer QLCs incorrectly in PV, are in danger of dropping from the course without learning programming (RQ2.1). Considering a university course, the consequences of dropping a course can be severe and it happens rarely in comparison to open online courses. In the university context of PVI, two out of three QLC types predicted lower course grades for students who answered incorrectly (RQ2.1). Next, I discuss explanations for the reduced course success.

The mixed methods research in PV suggests that students who repeatedly answer QLCs incorrectly have more programming challenges. According to intervention guidelines developed in PIV and presented in Section 6.3, many of those challenges became unproductive and the instructor should have been available to help the student forward. The students, who are weaker at QLCs, may solve challenges less systematically and resort
to tinkering behavior (RQ2.3).

These results suggest that QLCs have the potential to detect fragile knowledge and produce early warnings. Moreover, QLCs can complement program writing exercises, which may not reveal the same issues, especially when automated assessment is used. In the future, QLCs could be designed to measure mastery of selected concepts or to detect known misconceptions. PVI identifies potential patterns in incorrect answers and discusses the related programs to produce hypotheses of possible reasons for incorrect QLC answers. I include a few example programs that have been reformatted and renamed from Finnish to ensure anonymity.

Example A in Figure 8.2 presents a student’s program and QLCs, where the student’s answers are marked as crossed boxes. They are not entirely correct. The answer to Q1 is missing \( x \), which is used as an iterator in a for-statement. Q1 was answered by 291 students of whom 42 answered incorrectly. There is a small pattern as four students ignore a variable they have used as an iterator. If the generated Q1 would have included numbers as an option that could spur another identified pattern. Seven students ignore a variable they use to reference a list. Such errors may indicate that students could have a different understanding of the term variable than the instructors have attempted to teach in this context. Additionally, they could expect those variables to behave differently than when assigning literal values or ordinary expressions.

Also, the answer to Q2 in Example A is incorrect as it should be line 6, which can raise a ValueError, which would be caught on line 7 in the except-block. Q2 was answered by 243 students of whom 34 answered incorrectly. Altogether 33 students answered the line that begins the try-statement. In 24 cases the correct line is immediately after that. Example B in Figure 8.3 includes another one of these cases. Students could in some ways misinterpret words “execution” or “jump” and perhaps the question could be improved. Students could argue that execution can jump from the try-block to the except-block. However, lines inside the try-block are included in the options and students could also think that the error is somehow detected before executing those lines.

Example B in Figure 8.3 has another incorrect answer in Q3. Line 13 has a conditional statement that proceeds to divide with \( i \) only in the case that \( i \) is greater than zero. Student however answers that the line’s purpose is to ignore negative input. Considering only the line, it would avoid negative as well as zero value. However, the purpose in this context
Example A: Student’s program

```python
def rain():
    numbers=[]
    amount = input("Enter rainfall. End with -999.")
    while amount != "-999":
        try:
            amount = float(amount)
        except:
            amount = float(-1)
        numbers.append(amount)
        amount = input("Enter rainfall. End with -999.")
    if not numbers:
        average = 0
    else:
        without = [x for x in numbers if x >= 0]
        average = sum(without)/len(without)
    return average
```

Q1: Student’s answer is checked
Which of the following are variable names in the program?
☐ return ☐ x ☐ float ☐ amount ☐ n

Q2: Student’s answer is checked
From which line can program execution jump to line 7?
☐ 3 ☒ 5 ☐ 6

Q3: Student’s answer is checked
Which of the following best describes the purpose of line 10?
☒ Accepts new data
☐ Guards against division by zero
☐ Ignores negative input
☐ Is a condition for ending the program

Figure 8.2. Example A—Student’s work on the assignment in PVI, where two answers are incorrect: Q1 is missing variable x and Q2 should be 6
Example B: Student’s program

```python
1 def rain():
2    sum = 0
3    i = 0
4    row = 0
5    while row != -999:
6        try:
7            row = input("Enter rainfall. End with -999.")
8            row = float(row)
9            if row >= 0:
10                sum += row
11                i += 1
12        except:
13            ValueError
14        if i > 0:
15            average = sum/i
16        else:
17            average = 0
18    return average
```

Q1: Student’s answer is checked
Which of the following are variable names in the program?
- [ ] except
- [x] row
- [ ] magic
- [x] average
- [ ] print

Q2: Student’s answer is checked
From which line can program execution jump to line 12?
- [x] 6
- [ ] 7
- [ ] 8

Q3: Student’s answer is checked
Which of the following best describes the purpose of line 14?
- [ ] Accepts new data
- [ ] Guards against division by zero
- [x] Ignores negative input
- [ ] Is a condition for ending the program

Figure 8.3. Example B—Student’s work on the assignment in PVI, where two answers are incorrect: Q2 should be 7 or 8 and Q3 should be “Guards against division by zero”
Discussion

is not to ignore any input. Q3 was answered by 291 students of whom only 10 were incorrect. A variation of Q3 would ask for the purpose of line 7 in Example B. For a line, where input reads, in other words, accepts, new data from a user, two students select “Is a condition for ending the program”. A string on that input line describes how to end the program but the code on that line is only concerned with querying and retrieving any input. These answers adhere to the line’s content on the surface rather than understanding its purpose in the program context.

Example C in Figure 8.4 has line 6, which queries for data, accepts new data, and attempts to store it as float-type. The line 6 is enclosed in a condition that executes the line if the previous input is negative. Four students claim that such a line in such a context ignores negative input. In this case, the line’s context is considered without making a distinction to the individual line. Alternatively, students may understand the order of execution incorrectly.

<table>
<thead>
<tr>
<th>Example C: Excerpt from student’s program</th>
</tr>
</thead>
<tbody>
<tr>
<td>1        while value != -999:</td>
</tr>
<tr>
<td>2  if value &gt;= 0:</td>
</tr>
<tr>
<td>3    values.append(value)</td>
</tr>
<tr>
<td>4    value = float(input(&quot;Enter rainfall. End with -999.&quot;))</td>
</tr>
<tr>
<td>5  else:</td>
</tr>
<tr>
<td>6    value = float(input(&quot;Enter rainfall. End with -999.&quot;)))</td>
</tr>
</tbody>
</table>

Figure 8.4. Example C—Excerpt from student’s work on the assignment in PVI

These observations in PVI suggest that superficial reading of code, inability to recognize certain code elements, and inadequate understanding of execution are candidates to explain incorrect answers.
The results are synthesized to answer RQ2.

**RQ2** To what extent can QLCs probe students’ knowledge and program comprehension?

Both success on a course and the ability to systematically overcome challenges in developing programs correlate with performance on QLCs. This suggests that QLCs can probe students’ program comprehension. However, the current studies have not researched, if QLCs can be designed to assess different programming concepts or areas of comprehension.

### 8.4 QLCs Could Accompany Critical Use of AI in Programming

The effect of AI in introductory programming can and has been seen from at least two points of view. First, will students try to outsource programming to AI and complete courses without learning themselves? Second, how and when should students learn to use powerful AI to help in programming? Program comprehension has been suggested as one area that needs more focus when programmers are expected to read and evaluate more AI-generated code [8; 91]. This makes QLCs relevant for both attempts to control and embrace the use of AI in introductory programming.

While current state-of-the-art AI can answer QLCs above the level of the average novice student, AI still makes mistakes for unpredictable combinations of AI-generated programs and QLC types (RQ3.1). Many of the incorrect AI-generated answers contradict themselves in their explanation (RQ3.2). This has an interesting consequence. If students acquire or are presented with QLC answers from AI, they should read the explanation to identify correct answers. This turns into a learning experience for both correct and incorrect answers. Answers have rich explanations but they should be read critically to detect logical problems. For example, students could learn careful tracing from an AI answer. However, while studying AI answers they need to develop a critical expectation for AI-generated output.

Unexpectedly, many errors that AI produces in QLC answers share characteristics with errors that real students make (RQ3.2). AI can argue using illogical execution steps or interpret questions differently than they are stated. AI can also jump into an answer without conducting a sufficient
level of analysis. Such similarities could support using AI-generated data when actual student data is somehow limited or not yet available.

The results are synthesized to answer RQ3.

**RQ3** How AI answers different types of QLCs?

AI answers QLCs better than the average novice but can unexpectedly produce incorrect answers. Many errors share similarities with errors that humans make. The answers also regularly contradict themselves in the explanation. Asking students to evaluate QLC answers produced by an AI could support the critical use of AI in programming.

### 8.5 Limitations

This dissertation has several limitations that should be considered when interpreting the findings. I discuss the limitations here in the scale of the dissertation. See each publication from PI to PVII for discussions on threats to their validity.

The studies presented in this dissertation rely on data obtained from students’ responses to QLCs and their behavior in programming tasks. This granular data allows us to study and observe complete student populations. However, interviews or think-aloud studies with individuals would produce firsthand knowledge of students’ thoughts.

The studies were conducted in multiple learning contexts, including two university courses in different countries and an open online course for lifelong learners. While this adds to the generalizability of our results, the populations are extremely varied and could include phenomena that rarely occur in a class for computer science majors in a top university. Therefore, I avoid making conclusions on single questions or populations and present results as overall trends.

All in all, the dissertation only produces first experiences and evaluations of using QLCs in introductory programming as represented by the research questions. More research is required to design pedagogical use of QLCs.
8.6 Future Work

The design of questions is a complicated effort. The individual QLCs should be scrutinized, for example using think-aloud studies to confirm what exact knowledge and skills students use to answer and whether the feedback is helpful for them. Furthermore, established methods to assess the characteristics of the different questions exist. Depending on whether QLCs are used more as individual prompts for reflection or as an instrument to measure learning, their difficulty, and quality could be evaluated using the item response theory [36].

To analyze the possible effects of QLCs on student behavior and learning, they should be systematically integrated into the course design. When students regularly interact with QLCs, they could affect the perceived importance of program comprehension or train students to use their metacognitive skills. For example, students could improve their ability to self-explain new knowledge. Confirming such hypotheses probably requires longitudinal studies.

QLCs could be developed further to serve in different use cases. One potential area of research is tailoring exercises to detect misconceptions [78]. QLCs could be developed for this purpose, and feedback from such QLCs could be designed to provide evidence that refutes the detected misconception. Another research direction would be using QLCs for unfinished programs. An integrated development environment (IDE) could monitor and generate QLCs immediately when an interesting program structure appears. Using QLCs in this way or generally in large numbers makes user modeling [17] a topic of great interest. The system could collect information on the mastery of skills at the level of individual QLC types and prevent overly repetitive QLCs.

Technically, automated generation of QLCs extends to generating questions about any programs, not only those created by the learner. For example, peer reviews of code could apply questions about the program code to confirm that the reviewer reads and to a degree comprehends the program before they are allowed to write a review about it. A completely new learning activity could first ask the student to use an AI to generate a small program. After that, the student would receive a few QLCs which are already answered by the AI. Finally, the student must critically evaluate which answers are correct and which are incorrect.
This dissertation defined QLCs and presented them as a potential approach to developing automated assessments toward program comprehension. The idea is to pose personal questions to students that target the structure and the logic of the program code they created themselves. Furthermore, the dissertation discussed design decisions for an automated QLC generator and produced such systems for three different programming languages.

QLCs were evaluated in practice on two courses at universities in different countries as well as on an open online course for lifelong learners. The results indicate that as many as 20% of novice students may not have the ability to systematically reason how their own program code works, although they had passed the functional tests for the program. Furthermore, more than half of novice students fail to trace their functionally correct program code. The results support previous findings where novice students required several weeks to develop the cognitive capacity to reason about program code.

Performance on QLCs correlates with students’ programming ability and course success. PV suggests that students, who are weaker at QLCs, easily resort to tinkering and spend considerable time making no progress. Therefore, QLCs could provide early warnings about students, who need additional help to learn programming.

State-of-the-art AI tools can answer QLCs better than an average novice but they still lapse into errors for unexpected questions and programs, although the AI was able to generate the programs correctly from the exercise prompt. The incorrect answers include explanations that contradict themselves as well as errors that share similarities with human errors. In addition to answering QLCs, students could be asked to evaluate QLC answers from an AI. Identifying errors based on the explanation of the AI could support the critical use of AI-generated output in the future.
in any way, automated questions about the structure and behavior of a program can provide scalable, early practice in discussing program code, which is a critical skill in the programming industry.
References


 References


[80] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, and Andrea De Lucia. On the diffuseness and the impact


References


References


[125] Ahmad Taherkhani, Ari Korhonen, and Lauri Malmi. Recognizing algorithms using language constructs, software metrics and roles of vari-


[133] Anne Venables, Grace Tan, and Raymond Lister. A closer look at tracing, explaining and code writing skills in the novice programmer. In *Proceedings


Errata

Publication II

Figures 2 and 3, two histograms at the bottom left corner:

“N=29” and “N=43” should be “N=27” and “N=45”