Differentiated Assessments for Advanced Courses that Reveal Issues with Prerequisite Skills: a Design Investigation

Greg L. Nelson*
University of Washington
Paul G. Allen School of Computer Science & Engineering
Seattle, Washington
glnelson@uw.edu

Filip Strömbäck*
Linköping University
Department of Computer and Information Science
Linköping, Sweden
filip.stromback@liu.se

Ari Korhonen*
Aalto University
Department of Computer Science
Espoo, Finland
archie@cs.hut.fi

Marjahan Begum
Copenhagen Business School
Department of Management, Society and Communication
Copenhagen, Denmark
mbe.msc@cbs.dk

Ben Blamey
Uppsala University
Department of Information Technology
Uppsala, Sweden
ben.blamey@it.uu.se

Karen H. Jin
University of New Hampshire
Department of Applied Engineering and Sciences
Manchester, New Hampshire
karen.jin@unh.edu

Violetta Lonati
Università degli Studi di Milano
Department of Computer Science
Lab. CINI "Informatica & scuola"
Milan, Italy
lonati@di.unimi.it

Bonnie MacKellar
St John's University
Division of Computer Science, Math and Science
Queens, New York
mackellb@stjohns.edu

Mattia Monga
Università degli Studi di Milano
Department of Computer Science
Lab. CINI "Informatica & scuola"
Milan, Italy
mattia.monga@unimi.it

ABSTRACT

Computing learners may not master basic concepts, or forget them between courses or from infrequent use. Learners also often struggle with advanced computing courses, perhaps from weakness with prerequisite concepts. One underlying challenge for researchers and instructors is determining the reason why a learner gets an advanced question wrong. Was the wrong answer because the learner lacked prerequisite skills, has not mastered the advanced skill, or some combination of the two? We contribute a design investigation into how to create *differentiated* questions which diagnose prerequisite and advanced skills at the same time. We focused on tracing and related skills as prerequisites, and on advanced object-oriented programming, concurrency, algorithm and data structures as the advanced skills. We conducted an inductive qualitative analysis of existing assessment questions from instructors and from a concept inventory with a validity argument (the Basic Data Structures Inventory). We found dependencies on a variety of prerequisite knowledge and mixed potential for diagnosing difficulties with prerequisites. Inspired by this analysis, we developed examples of differentiated assessments and reflected on design principles for creating/modifying assessments to better assess both advanced and prerequisite skills. Our example differentiated assessment questions and methods help enable research into how prerequisites skills affect learning of advanced concepts. They also may help instructors better understand and help learners with varying prerequisite knowledge, which may improve equity of learning outcomes. Our work also raises theoretical questions about what assessments really assess and how separate advanced topics and prerequisite skills are.

*Leaders

CCS CONCEPTS

• **Applied computing** → **Education**; • **Theory of computation** → *Concurrency*; *Design and analysis of algorithms*; • **Information systems** → *Data structures.*

KEYWORDS

computer science education, assessment, prerequisite skills, concurrency, data structures and algorithms, advanced object-oriented programming, tracing, educational design research, differentiated assessment

ACM Reference Format:
Greg L. Nelson, Filip Strömbäck, Ari Korhonen, Marjahan Begum, Ben Blamey, Karen H. Jin, Violetta Lonati, Bonnie MacKellar, and Mattia Monga. 2020. Differentiated Assessments for Advanced Courses that Reveal Issues with Prerequisite Skills: a Design Investigation . In *2020 ITiCSE Working Group Reports (ITiCSE-WGR '20), June 17–18, 2020, Trondheim, Norway.* ACM, New York, NY, USA, 55 pages. https://doi.org/10.1145/3437800.3439204

1 INTRODUCTION

In computing education research (CER), we have initial evidence that, when learners take advanced classes they have weaknesses with prerequisite knowledge, which may negatively affect learning outcomes. A 2004 ITiCSE working group [44] suggested that many students lack mastery of program tracing after CS1, which contributes to their poor problem solving skills. Other studies show this may continue into later courses. Valstar et al. showed that more than 30% of students could not do questions on pointers or tracing recursion at the start of an upper level data structures class, which correlated with final exam scores [89]. Fisler et al. showed more than 30% of 3rd and 4th year CS majors failed questions on scope, parameter mutation, and/or variable mutation, suggesting knowledge transfer requires explicit instruction and/or reinforcement of prerequisites [14].

Clearly, assessments for advanced courses aim at assessing specific advanced topics, even though they often require prerequisite knowledge or skills. In general, the cause for a learner to fail such assessments may be related to 1) difficulties with the advanced concept or 2) weaknesses with basic concepts that may lead to incorrect application of the advanced concept. However, assessments for advanced topics are not intended to assess prerequisites explicitly, and may not be suited to support the right diagnosis.

For example, concept inventories for advanced topics seem unlikely to already diagnose problems with prerequisites, for two reasons. First, while concept inventories for advanced topics include distractors for misconceptions, they are designed by definition to cover misconceptions about the advanced concepts, not the prerequisites. Second, most concept inventories are actually never evaluated for how well they can diagnose student thinking or give formative feedback to students; just because they measure knowledge does not imply that measurement is useful for giving feedback [11, 28, 55, 73, 74]. To our knowledge there has been no analysis in computing education of whether the distractors for a concept inventory might diagnose prerequisites.

Pre-exams are also a possible tool to identify issues with prerequisite at the beginning of an advanced course. The drawback is that both administering pre-exams and addressing the identified issues is time consuming, and may also result in "boring" activities for students. Moreover pre-exams may not be enough to guarantee that the prerequisite skills are at the level required to focus properly on the new ones to be learned.

Finally, advanced topic assessments asking learners to "show your work" can sometimes diagnose prerequisite issues but require lengthy manual interpretation. Instructors can give advanced questions and ask students to "show your work", an ancient and useful technique, but it takes a lot of time and instructor expertise to grade and write feedback.

In response to these drawbacks, our working group contributes a new genre of assessment: **differentiated assessments**, which are advanced topic assessment questions that also can diagnose relevant prerequisite issues. The key idea is to expand the typical scope we define for an assessment, to cover more of prerequisites. Unlike pre-exams, differentiated questions would be easier for advanced course instructors to use because they serve both purposes. Ideally, different incorrect answers would map to difficulties with the advanced content and difficulties with prerequisites, and also be easy to grade or even generate feedback for automatically.

The goal of our working group is to conduct a preliminary investigation into whether it is possible to design differentiated assessments, and if so, how this might be achieved. More precisely, we seek to contribute methods for both highlighting which prerequisite skills are measured by existing assessments, and to revise advanced assessments to make them more differentiated.

Differentiated assessments could be incorporated into advanced courses, and help instructors give better formative feedback (and potentially improve instructional design of courses by including observed frequently weak prerequisite skills). The scope of prerequisite knowledge investigated by differentiated assessments is necessarily limited, but can more narrowly focused on what is actually instrumental to the advanced topics. This might also help students in making sense of what they learn. The more students come in with varied backgrounds, especially in more advanced degree programs, the more generic prerequisite knowledge assessments can be problematic: having something clearly connected to the learning objectives seems a good way of supporting student diversity and equity.

The distinction between prerequisite and advanced skills clearly depends on the course context; prerequisite skills are those skills that students are assumed to be familiar with before attending a course, which are different from the new concepts the course aims to teach. For example, as a student progresses through their education, topics that were once new and difficult increasingly turn into prerequisites for other topics in other courses. For the purposes of our study, we consider as *prerequisite* skills or knowledge those that pertain to basic programming, and that are typically expected as learning outcomes of CS1. In particular we focus on the syntactical and conceptual knowledge of basic programming constructs, and on program comprehension skills including code tracing. While there are many other and important prerequisite skills one might consider, such as mathematics or basic literacy and writing skills, we did not include those in our scope.

As advanced topics, that in the report will be referred to as *course topics*, we chose algorithms and data structures, advanced object-oriented programming, and concurrency, as these are comparatively well-studied in CER, and match the expertise and teaching experiences of the working group members. In particular, we also considered in our study questions from the Basic Data Structures Inventory (BDSI), a concept inventory for data structures [66], which we chose because it is a recent, state of the art assessment with a validity argument supported by empirical studies.

To aid our assessment design investigation, we asked the following research questions:

RQ1: What prerequisite skills do advanced CS questions depend on?
RQ2: To what extent can an existing concept inventory for data structures with a validity argument – the BDSI [66] – also diagnose difficulties with prerequisite skills?
RQ3: What are examples of differentiated assessments and principles for designing differentiated assessments?

It is worth noticing that the RQ3 actually comprises two subquestions (concerning examples of assessments, and principles for designing assessments); however we merged them together because they are strongly connected and the methods we use to address them are interleaved, as will be presented in Section 3.
By answering these three research questions, we contribute:

- An **inductive qualitative analysis** of a selection of existing assessment questions for advanced courses, showing which prerequisite skills they require;
- A **code book** with prerequisite topics and concepts from the qualitative analysis;
- New **differentiated** assessment question designs that make implicit prerequisite assessment more explicit, so we can tell when the incorrect answers are caused by inadequate prerequisite skills;
- A **collection of design principles and strategies**, PAPRIDA (PAtterns and PRinciples for Differentiated Assessment) for modifying existing questions for advanced course topics to include more explicit assessment of prerequisite skills.

Empirical validity evidence that PAPRIDA and the examples of differentiated assessments work in practice are left to future work.

The remainder of this paper is organized as follows. In Section 2 we present related work. Next, in Section 3, we present the method used in the paper. The results are presented Sections 4 to 6, according to the three research questions. In particular, in Section 4 we present the results from our investigation on how prerequisite skills are present in existing assessments, answering RQ1. In Section 5, we explore how well the BDSI is able to differentiate between prerequisites and course topics answering RQ2. Lastly, in Section 6 we present the questions modified while developing PAPRIDA, and the patterns and principles in PAPRIDA itself, answering RQ3. Finally, we present some limitations with our approach in Section 7, discuss the results in Section 8 along with potential future work, and draw some conclusions in Section 9.

2 RELATED WORK

In this section we first consider skills related to basic programming knowledge, that we consider here as prerequisite skills; in particular in Section 2.1 we present three comprehensive works that identify CS1 knowledge, then we focus on tracing skills in Section 2.2. In Section 2.3 we present related works concerning the three advanced courses considered in the report. Work on assessment design is presented in Section 2.4.

2.1 Basic Programming Knowledge and Skills

In order to determine concepts that are prerequisites, it is important to understand the space of computer science concepts taught in typical undergraduate programs, and to determine which concepts are considered to build on other concepts. There have been some attempts at enumerating and ordering the standard concepts taught in CS undergraduate programs. Three important references are presented here that identify basic programming knowledge: the ACM 2013 Curriculum Guide [27], Goldman at al's list of core concepts taught in CS1 [17, 18], and Sorva's misconceptions catalogue [81]. Our group used these three lists as comparison standards for our list of identified concepts. A table mapping our concepts to the concepts from these three sources can be seen in Section 4.1, Tables 1 to 6.

ACM CC2013. The ACM 2013 Curriculum Guide [27] consists of an enumeration of topics that correspond to a wide set of Knowledge Areas (KAs) in computing. Its explicit purpose is to serve as a standard for computer science programs. It is developed by a task force over a two year period and subject to extensive external review. As a result, the ACM Curriculum Guide is widely accepted as a benchmark by universities developing computer science programs, which means its set of KAs and associated topics match the content of particular courses or course sequences in many programs. Thus, it is a useful catalog of computer science topics that are commonly taught in undergraduate computer science programs. Because of its comprehensiveness and wide acceptance, our group chose to use it as a basis of comparison with the topics that we identified during the process of analyzing existing assignments (see Section 3).

KAs are divided into knowledge units, and ultimately into topics for each knowledge unit (See Appendix A.1). The guide frequently identifies which knowledge units are advanced and which are introductory. In particular, the Software Development Fundamentals KS (SDF) is described as foundational to the other software oriented KAs. Since we are focusing on prerequisite skills in programming courses, Software Development Fundamentals (SDF) and Programming Languages (PL) KAs are the most relevant to our study. Since the advanced courses we included in our study are data structures, advanced object-oriented programming and concurrency, the following topics are also particularly important: SDF/Fundamental Programming Concepts, SDF/Fundamental Data Structures, and SDF/Development Methods as well as PL/Object-Oriented Programming, and PL/Basic Type Systems. The topics are described at a very a high level of abstraction. The related skills and knowledge are not based on any particular programming language; CC2013 refers to these essential skills as a *lingua franca* in which other computer science concepts can be described.

Core concepts identified by experts. Goldman et al. [17, 18] set out to create a concept inventory for introductory computing subjects. An important part of this process was to investigate which core concepts are typically covered in introductory courses, and which of those are perceived to be important and difficult. This investigation was done by consulting experts (i.e., instructors) using a Delphi process. Three panels were assembled [18], one for each of the three subjects examined: programming fundamentals, discrete math and logic design. Each panel consisted of around 20 experts that were either instructors or authors of learning material in the subject. The authors concluded that the Delphi process was useful for reaching consensus; there were, however, some concepts where a good consensus was not reached. Since the concepts listed in this paper are specifically identified as introductory, it is likely they are typically prerequisites to more advanced courses. For this reason, we considered this list as a possible starting point when analyzing existing assessment (see Section 3).

A summary of the final concepts for programming fundamentals is reported in Appendix A.2.

Misconception catalogue. Misconceptions can be caused by a lack of knowledge of how a particular syntactical construct behaves, i.e., due to an incorrect or incomplete understanding of the notional machine [13, 20].

Several misconceptions in introductory programming have been identified and addressed in the literature [38, 61, 67, 82]. For example, a classic misconception is the use of an assignment operator (=) instead of the comparison operator (==); another example of a misconception is that a variable can hold more than one value, this is manifested in the task of swapping two variables. Sorva collected a wide catalogue of misconception in [81], grouped under 11 topics (See Appendix A.3). Although it gives examples of novice programmers' misconceptions about the content of introductory programming courses in general, it leans towards misconceptions found in courses taking the object-oriented approach. In this research misconceptions relevant to the advanced courses are highlighted in Section 2.3.

2.2 Tracing Skills

Although introductory programming education traditionally focuses on writing programs, there is a large body of research on reading, tracing and understanding programs, suggesting tracing knowledge is critical for learning writing skills [26, 46]. The relationship between student skill of reading, writing, tracing, and explaining, as identified by assessments, is well studied in the BRACElet project, reported in [6], that originate after an ITiCSE working group in 2004 [44]. In particular the project started the 'Explain in plain English" style of questions and used Bloom and SOLO taxonomies to analyse them [91].

Tracing can be defined as the *step by step execution* of code, following the control flow path. This can be done with pencil-and-paper, mentally, using a debugger, or with the help of specific interactive visualization tools. Tracing tasks can include, for example, predicting the output of a segment of code, or making diagrams representing the contents of memory after some code executes. The report of an 2019 ITICSE working group also include a catalog [26, Section 5] of program comprehension tasks, covering many different kinds of tracing tasks.

Tracing requires the application of syntactic and semantic knowledge implied by the fragment of code machine, i.e., the notional machine [13]. For this reason tracing activities are considered to have an important role in understanding the notional machine [24, 57]. Tracing tasks are also suitable to unveil learners' shortfall or misconceptions, either related to basic semantic knowledge or concerning more advanced concepts as scope, references, parameter passing, that have major impact in later courses. On the other hand, errors in tracing tasks may also occur when the execution simulation is not performed rigorously or systematically enough. For instance, when debugging their own programs, learners may be influenced by what they *want* the program to do instead of identifying the exact actions prescribed by the code [63].

Xie et al. show that explicitly teaching tracing strategies can improve novices performances [93] and present a pedagogical approach where tracing acts as the first of four skills that the novice should develop [92].

Tracing activities can also be part of program comprehension strategies, including building strategies at higher levels of abstraction. For instance, executing a piece of code on a series of specific inputs may guide the learner to understand what the given code computes [26]. This high-level skill also includes understanding which fragments of code to be traced, or knowing when an external representation (e.g., a tracing table or a diagram) is needed.

Tracing can be shown by educational software tools that visualize step-by-step code execution or support visual program simulation exercises, see [83] for a survey. Such tools often present a simplified view or alternative view than that available from debuggers in modern IDEs, which allow line by line execution of a program whilst inspecting variables and the call stack.

They often support automatically assessed exercises and other interactive learning materials. These are important resources for large courses that might have hundred or even thousands of students. For example, PLTutor [57] integrates instruction on PL semantics, tracing, showing program execution, and integrating interactive practice questions, targeting CS1 level skills and recursion. Other examples include Python Tutor [19] which visualizes the execution of Python programs, and UUhistle [84] that additionally provides interactive puzzles in which learners use graphical controls to direct a program's execution. Students are supposed to learn several skills – including tracing skills – better than by simply watching an animation of execution of a program [54].

2.3 Advanced Courses and Topics

As mentioned in the introduction, in this report we consider three *course topics*: advanced object-oriented programming, data structures and algorithms, and concurrency.

We will discuss related work focusing in particular on the relation between advanced topics and prerequisite skills, including tracing. To the best of our knowledge, however, there is no work that specifically aims at identifying ways to differentiate issues with prerequisite and course topics.

2.3.1 Advanced Object-Oriented Programming. Although Object-Oriented Programming (OOP) is sometimes presented also in introductory courses, most of the techniques used by object-oriented approaches to structure and reuse large code bases are left to advanced courses. In this paper we try to be agnostic of any particular language used at the introductory level, but we assume the topics a learner is supposed to master after a programming course taught by referring to an object-oriented language include classes, objects, methods, etc. Peter Wegner famously defined OOP as a paradigm in which programs deal with classes, objects, and inheritance [90]. But even by limiting the approach to class/object encapsulation, many misconceptions related to OOP topics have been studied [25, 29, 68, 71, 80], such as:

- the problems with handling references to primitive types and user defined ones;
- classes and objects (e.g., confusion between a class and its instances, classes as collections of objects instead of templates for creating them);
- object identity and immutability;
- object state (e.g., that an object can only hold instance variables of a single type);

- methods (e.g., a method can be invoked on an object only once);
- the control flow among objects.

Advanced users of OOP put emphasis on modularity issues and the potential that a fragment of code designed for a specific goal might be re-used in a different context [43, 53, 58]. To this end, it is important to appreciate polymorphic types and late binding, together with the constraints posed by Liskov's substitution principle. The difficulties of dealing with a mix of static and dynamic typing, method dispatching, dependencies on abstract interfaces and concrete implementations, should be distinguished from more basic difficulties with the fundamental programming skills.

Cross et al. [10] found a positive student response to the use of a debugger in the context of teaching introductory programming with Java. Their particular teaching case concerned stepping through constructors in the inheritance chain when teaching OOP.

2.3.2 *Data Structures and Algorithms.*

The second topic covered in this paper is that of Data Structures and Algorithms (DSA), which is one of the key learning goals in some CS2 courses [65]. An introductory data structures course typically covers classic structures – stacks, queues, hash tables and binary trees, as well as learning about abstract data types and the application of these structures, while advanced data structure courses introduce topics on balanced search trees, graphs and sorting algorithms. Even though there have been continuous calls to place more emphasis on teaching how to apply data structures in larger programs, most existing assessments still focus on implementation [78].

Students entering advanced DSA courses were found to have low proficiency on prerequisite knowledge. Valstar, Griswold and Porter [89] administered a test consisting of questions on prerequisite topics to students starting a course in advanced data structures. Each question was designed to test proficiency on one prerequisite topic. The topics ranged from basic topics like pass by reference/value to topics from the prior data structures course such as list implementations and runtime analysis, as well as topics from a computer organization class. They found that students were not very proficient on many of the prerequisite topics, and that weakness on prerequisites correlated with poor performance on the final. In addition, they looked at which questions, when answered correctly, correlated with good performance on the final; and which questions, answered incorrectly, correlated most strongly with poor performance on the final. The contribution of Valstar et al.'s paper is to not only show that poor prerequisite knowledge corresponds to poor performance in a course, but to give a more detailed analysis of the particular topics most correlated to course performance, which was extended upon in a paper the next year [37].

Visual program simulation exercises can be utilized in advanced courses to recap and evaluate if students have misconceptions on prerequisite skills. For example, UUhistle has been utilized in DSA to teach recursion (how stack frames are formed), but also for catching possible misconceptions students have related to the basic notional machine [84]. The OpenDSA project [15, 75] provides a complete interactive text book for DSA that includes a number of proficiency exercises that ask the student to simulate an algorithm step by step. This kind of visual algorithm simulation [36] resembles the process of tracing a fragment of code, but it is done in a much higher level of abstraction than in visual program simulation.

The relation between writing, tracing, and reading skills in CS2 have been investigated in [8, 22, 62] with conflicting results.

2.3.3 *Concurrency.*

The third topic we will cover in this paper is concurrency. There are several models of concurrency that are being taught and used. In this paper, we will assume the model where multiple threads execute in a single process and communicate using shared memory and synchronization primitives such as semaphores, locks and condition variables. In this model, the programmer needs to make sure that important steps happen in the right order, for example waiting for the completion of another thread, and to ensure that multiple threads do not access the same piece of data concurrently, usually using locks. This model has previously been explored in various education contexts by a number of authors. One such example is Kolikant, who studied high-school students' solutions to synchronization problems involving semaphores [34] in a concurrency simulator proposed by Ben-Ari et al. [64]. From the solutions, the author found that the problematic part was to identify the synchronization goal. Once the goals were identified, most students managed to solve the problem. In a later study involving questionnaires [35], the same authors investigate students' views on what properties a concurrent program should have to be considered correct, and found that many students considered a program that exhibited occasional crashes or errors due to concurrency to be correct, which might impact their performance on concurrency questions.

Another study of common mistakes in this model of concurrency was made by Strömbäck et al. [86], who examined a large number of student solutions to a concurrency problem involving synchronizing a simple data structure. The results differed slightly from what Kolikant found. While 89% of students managed to identify an instance where a busy-wait loop needed to be replaced by a semaphore, only 62% managed to do so properly. A similar trend was observed when students were asked to synchronize shared data using locks. While a majority of students identified at least some aspects of the issue, only around half managed to arrive at a correct solution using locks. In this process, the authors identified a number of mistakes made by students. Some of these are likely due to students not understanding how the synchronization primitives, mainly semaphores and locks, work. The authors do, however, note that some of the mistakes are possibly due to lacking prerequisite skills, such as lacking tracing skills and a lacking understanding of pointers and scoping of variables.

Other concurrency models have been studied in an educational context. For example, tuple spaces [5] which have been studied extensively by Lönnberg et al. [45, 49], and the actor model [12]. The actor model is similar in nature to real-world events and it has been studied in this context as well [33, 42]. Moreover, Lönnberg et al. have studied how to utilize these models to study students approaches to debug concurrent programs. In their study, students make use of Atropos [45, 49], a novel visualisation system targeted to debugging concurrent programs. The tool is intended to display information relevant to understanding the behaviour of concurrent Java programs, and is capable of visualising a trace of concurrent program for post-mortem analysis in the form of dependence graphs.

2.4 Assessment Design

In the computing education literature, assessment is a frequently studied topic; most works however focus on the assessment of either basics/prerequisite or advanced skills. Our *differentiated* assessment questions try to do both. For a review of works on assessment for introductory programming, the reader may refer to [39, 48].

2.4.1 Diagnostic Assessments and Concept Inventories. In the educational assessment field, assessments that support very specific inferences about different parts of a learner's knowledge have been called diagnostic assessments [41]. Concept inventory (CI) tests are assessments with validity arguments for assessing the overall understanding of a student for a particular set of concepts [1, 70, 87]. They are characterized by being criterion-referenced (rather than norm-referenced), and are often presented in the form of multiple-choice tests, including "distractors" that cover frequent misconceptions. As a form of assessment, either might be used either formatively, perhaps at the start of a course in an effort to assess prior knowledge, or summatively. Diagnostic assessments and concept inventories have subtle but important differences.

While it is a frequent perception among researchers that concept inventories can diagnose a particular test-taker's misconceptions, research in educational assessment has problematized that assumption. Jorion et al. shows that validity arguments need to be made separately for diagnosing test takers, and shows the different studies and analyses needed for such claims [28]. Sands et al. includes discussion of the same issues in physics around the original Force Concept Inventory and in concept inventories in general [73]. Santiago Roman's dissertation and Denick et al. are examples of trying to interpret a CI more diagnostically [11, 74].

2.4.2 Assessments in Computing. Prior work on assessment has identified issues with assessment questions covering too many skills, which inhibits inferring why the learner got the question wrong when trying to give feedback or help them learn. The 2017 ITiCSE working group on assessing programming fundamentals showed how a single question can assess many skills/parts of knowledge [47]; for example, small errors, like not understanding modulus operators, can cause learners to get large questions incorrect [32]. In our analysis of assessment we follow a similar approach, aiming at identifying the many prerequisite skills required.

Several assessments use code tracing questions in order to assess programming knowledge; others assess tracing skills themselves, as the already mentioned [26, 44, 47]. Nelson et al. developed a systematic, multi-part formative assessment for tracing skills of simple statements as well as larger combinations of code [55].

Within computing education for CS1 knowledge, several assessments and concept inventories have been developed. For example, Parker, Guzdial, and Engleman [59] designed the SCS1, a concept inventory for introductory programming concepts (including fundamentals, logical operators, conditionals, iteration, arrays, parameters, recursion, and object-oriented basics). Other work on developing standardized assessments of introductory computer science concepts include Caceffo et al. [4], who developed a concept inventory for a C programming course including parameters and functions, iteration, loops, variable scope, recursion, pointers, and structures; Simon et al. [76, 77] who developed a set of benchmark questions to be embedded in exams in introductory programming courses at a range of institutions; Snow et al. [79], who used Evidence-Centered Design to develop a set of assessments for high school students taking the AP CS Principles course.

Work has also been done on assessing advanced topics. For example, assessments have been developed for algorithms and data structures [31, 60]. Assessments with explicit validity arguments have been developed for recursion [21]. The Basic Data Structures Inventory (BDSI) has the most extensive empirical work, which covers some basic data structures and algorithm runtime analysis concepts, and was developed as a concept inventory [66]. Question banks also contain instructor-submitted questions that cover advanced topics [16, 72]. Moreover, the OpenDSA project [15, 75] that was mentioned earlier as well as visual algorithm simulation exercises in general [36] provide a meaningful way to assess and give immediate feedback for students solving tracing exercises on data structures.

To the best of our knowledge there are no assessments designed to assess simultaneously prerequisites and advanced skills, in a way that it is possible to differentiate among them. Even concept inventories for advanced topics, although with validity arguments for measuring advanced topics, may not be suited to detect prerequisite issues, since 1) they aim at assessing advanced topic knowledge and 2) their distractors are, by design, aimed at detecting misconceptions about the advanced topics.

In our study we borrow from concept inventories the idea of using distractors related to misconceptions; in our case the goal is using some of the distractors to identify prerequisite issues.

2.4.3 Assessment Validity. Tew and Dorn argue for importance of assessment validity in computing education [88], and Nelson et al. present Kane's validity framework for a computing education audience [55]. Assessment validity concerns trace back historically to concerns in psychology about whether some survey or other psychological measurement technique actually measures what it purports to measure [30, p. 6–7][9], and epistemological issues around what observations provide evidence relevant for theoretical statements (for example, disproving a theory) [85]. Validity arguments for a particular (pragmatic) use of an assessment draw on pragmatism [40] and the ancient human wisdom of asking "Why? For what purpose?". While our paper recognizes the importance of making strong validity arguments for assessment, our paper focuses on trying to create new designs for assessment questions that can differentiate between issues with prerequisite and advanced course topics, and does not aim at contributing strong validity evidence that our designs work in practice.

3 METHOD

The goal of our work is to explore the issue that assessment for advanced courses often implicitly assess prerequisites, and to design improvements to these assessments to make prerequisites explicitly assessed. As such, we are conducting *research through design*, exploring a problem by designing possible solutions to the problem and analyzing them. As is described by Herriott [23], design through research can be thought of as conducting an experiment to test a hypothesis. In this case, the hypothesis is that it is possible to address prerequisites explicitly together with course topics by making differentiated questions, and we are testing this hypothesis by designing such questions. As a side-effect of this inquiry, we will also propose methods for discovering hidden prerequisites in assessments, and for assessing them explicitly. Herriott also notes that design through research is well suited to conducting a systematic inquiry into a subject to discover new knowledge about it. As such, our contributions are mostly related to exploring the problem and designing possible solutions rather than providing empirical evidence for the validity and/or generalizability of these solutions. Therefore, we have adopted the ideas in *educational design research* suggested by McKenney and Reeves [50, 51]. According to the authors, "educational design research is a genre of research in which the iterative development of solutions to complex educational problems provides the setting for scientific inquiry," [50] and attempts to solve real-world problems while seeking to discover new knowledge that might be valuable to others facing a similar problem. Since educational design research is not a single method, but rather a way of approaching a problem, there is no well-defined procedure to follow. McKenney and Reeves do, however, describe the following characteristics of educational design research [50]:

- It uses scientific knowledge, and also other kinds of knowledge, such as craft wisdom, to ground design work.
- It produces scientific knowledge, and also craft wisdom among the participants in some cases.
- It strives to develop both interventions and reusable knowledge.
- It is an iterative method. In our case, we follow the method suggested by Reeves [69], which consists of four phases: *problem analysis*, *solution development*, *iterative refinement*, and *reflection to produce design principles*.

In educational design research, the different phases are often revisited during a project, and the results are thus refined during the project. According to the authors, it is essential to involve practitioners in this kind of educational design research in order to properly identify real teaching and learning problems, and to create prototype solutions based on existing principles and prior experience. The participating researchers in the current study are active faculty members and/or active researchers in computing education research with primary responsibility for not only teaching their courses, but also designing the courses and developing course assessments. Together, the researchers represent over 100 years of collective experience in pedagogy and course design. In addition, several of the researchers have had significant responsibility for curriculum design at the program level, in official capacities such as CS program director, service on departmental curriculum committees, and through development of new programs at their universities. As a result, they brought to this task experience in course development at both the course and the program level, as well as years of experience developing course materials and assessments.

Our design process consisted of the four steps that are outlined in Figure 1:

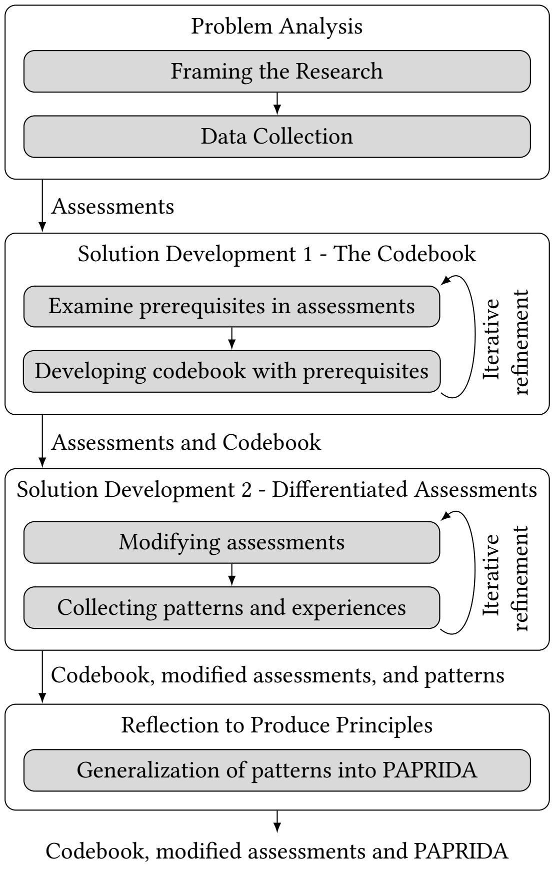

Problem Analysis
Framing the Research
Data Collection
Assessments
Solution Development 1 - The Codebook
Examine prerequisites in assessments
Developing codebook with prerequisites
Iterative refinement
Assessments and Codebook
Solution Development 2 - Differentiated Assessments
Modifying assessments
Collecting patterns and experiences
Iterative refinement
Codebook, modified assessments, and patterns
Reflection to Produce Principles
Generalization of patterns into PAPRIDA
Codebook, modified assessments and PAPRIDA

Figure 1: Overview of how the educational design research method was applied.

Problem Analysis and Data Collection: This phase includes framing the problem, which is described in the introduction, and collecting data to analyze further.

Solution Development 1 - The Codebook: We identified and coded prerequisite skills required by each question in each assessment. A codebook for prerequisite skills was simultaneously developed throughout this process, taking inspiration from prior skill classifications. As we will present in further detail below, *iterative refinement* is an important part of this phase.

Solution Development 2 - Differentiated Assessments: We revised a subset of the questions in the assessments to make them able to differentiate between weak prerequisite skills or weak knowledge of course topics. Once again, *iterative refinement* is an important part of this phase.

Reflection to Produce Principles - PAPRIDA: Intertwined with the *solution development 2* phase where we created differentiated assessments, we also identified and refined patterns and principles for designing such assessments.

The methods used in each step are detailed in the following subsections:

3.1 Problem Analysis and Data Collection

We decided from the outset to limit the scope of assessment material considered to “core” programming-related problems, including those related to pseudo-code. We excluded courses related to, for example, mathematics, and more applied or qualitative topics within computer science (such as security, ethics, software engineering, databases, etc.).

Assessments were contributed by the authors from undergraduate and masters level courses from a range of institutions in Europe and the United States. We specifically selected assessments covering the topics of concurrency, algorithms and data-structures, advanced object-oriented programming. The rationale being that these topics are widely taught, but are normally outside the scope of CS1 courses, and build upon some parts of CS1. As previously mentioned, we refer to these as *course topics*, implying that they are what is intended to be covered, compared to *prerequisites*. We also selected assessments which were known to be problematic for students due to lacking prerequisite skills. This selection was based on the members’ collective experience in teaching these topics. Some members even created pretests specifically constructed to address prerequisite knowledge. In these cases, the pretest questions were also included in the set of questions, as they may act both as inspiration for how to better assess prerequisite skills in assessments for course topics, and that they may be used as a foundation to build new such questions on top of. We also included the BDSI [66] as a source for questions, to answer RQ2 to see what prerequisites are involved in a state of the art concept inventory. In total, we looked at 11 questions from courses taught by the authors, and all 13 questions from the BDSI. The 11 questions from courses taught by the authors, as well as two questions from the BDSI are available in Appendices M and O. Questions labeled M.x are questions that were later modified, questions labeled O.x are questions that were analyzed but not modified, and questions labeled B.x are questions from the BDSI.

The selected assessment materials were a mixture of “pen and paper” exercises, and those requiring writing new code on a computer. With our selection of programming-related assessment material, tracing skills were a clear prerequisite, and a natural way of framing our approach. Furthermore, tracing was a way to frame our thinking about the assessment material, so that we were able to think critically about implicit prerequisite knowledge (“what does a student need to know about to trace this code?”), as well as a means of devising new questions (“where in the code do you see...?”, “what happens when this code is executed...?”)

3.2 Solution Development 1 - The Codebook

The goal of the qualitative analysis is to examine which, if any, prerequisites a student needs to understand in order to answer each question properly. We first considered a deductive approach based on a predefined codebook. To find a theoretical or a authoritative source for this codebook, we considered using the ACM 2013 Curriculum Guide [2], the Core Concepts identified by Goldman et al. [17, 18], and the Misconception Catalogue compiled by Sorva [81] (See Appendix A). Each of these potential sources were, however, not deemed suitable for our analysis. While the ACM 2013 Curriculum Guide is very complete, the main problem for our usage is that a number of topics are described at a high level of abstraction, while we desire a higher level of detail. For example, the question O.7 (available in the Appendix) assesses whether a student has realized that the `return` statement halts execution of the current function in addition to defining what value to return from the function. This skill in particular maps to the ACM Curriculum Guide concept of “SDF/Fundamental Programming Concepts/Functions and parameter passing”. This concept does, however, also cover the aforementioned “what to return” and function parameters in addition to halting execution of the current function. The main issue with Goldman’s Core Concepts is that it is only intended to cover introductory concepts. As the specific prerequisite concept for a course varies depending on the level of the course and what is covered in earlier courses, the topics covered by the Core Concepts might not cover all potential prerequisites for our questions. The previously mentioned example with the return statement illustrates this issue as well: the Core Concepts does not include a concept that involves return, neither the return value nor the act of halting execution. Finally, the Misconception Catalogue would indeed form a detailed codebook, but due to its high focus on object-oriented programming, it was also deemed unsuitable. The example regarding the return statement once again illustrates this: it lists misconceptions regarding the return value, but not regarding the act of halting execution.

Since none of the above-mentioned candidates were deemed suitable, we opted for an inductive coding approach, and hence built a new codebook. This will give us the additional benefits of a bottom-up approach: the codebook will be representative of what the questions contain, regardless of the completeness and level of detail of another source, and allows our codebook to focus on skills related to tracing, and allows the code to represent prerequisites at a sufficient level of detail. This is particularly important in this context, as the relation between prerequisites and course topics in assessments are not well studied. We do, however, recognize the importance of the above-mentioned works, and we will therefore relate the codes created in this work to them. Thus, that relation can be used to prioritize prerequisites to assess based on their location in the ACM 2013 Curriculum Guide, their importance in the Core Concepts or known misconceptions.

The analysis of the assessments and the development of the codebook were done iteratively. First, researchers analyzed the prerequisites assessed in each of the questions in each of the assessments, and then produced a proposed coding for these. The produced codes were then discussed with the group, and the two previous steps were revisited (which constitutes the iterative refinement phase).

In order to account for the different experiences and viewpoints of the coders, the working group members were divided into two groups, and each assignment was assigned to at least one member in each group (thus, each question was assigned to at least two coders). Each coder then independently coded the prerequisites required to answer each question in each assessment. The analysis and development phase was iterated a number of times within each group before a consensus was reached.

When both groups were done with their coding work, the two groups met and discussed their findings, aiming to merge the codes created by the two groups. This corresponds to another design phase. After this, the two groups independently re-coded all questions using the unified codebook. This resulted in a number of minor revisions to the codes, which were then discussed between the two groups again. This was repeated until all members of the working group were satisfied that the codes covered all prerequisites covered by the collected assessments. Note that due to the iterative refinement of the codebook, some codes were split into two during the process, resulting in codes representing prerequisites that are not assessed by any of the questions. As these still represent valid prerequisites, they were kept in the codebook in spite of the fact that they are not present in any of the questions. The codebook, and our coding of the assessments are presented in Section 4.

In order to examine whether the questions in the BDSI had similar issues to the other assessment we collected, we examined the distractors in the BDSI once more after the codebook was finalized. For each question, two researchers independently coded which prerequisite skills, if any, a student picking each distractor might have difficulties with. In order to properly distinguish between prerequisites and course topics, the researchers also noted which course topics a student could have difficulties with. As with the other coding, the researchers then discussed any potential differences until agreement. With this information we can see which prerequisite skills are assessed by each question in the BDSI, and to what extent it is able to differentiate between difficulties with prerequisites compared to course topics. The results from this coding is presented in Section 5.

Finally, in order to verify that our codebook was sensible, and what areas are covered by the other sources presented previously, we mapped the codes created in this stage to the ACM 2013 Curriculum Guide, the Core Concepts and the Misconception Catalogue. For the first two, this mapping was also done independently by two researchers in a similar fashion to the coding. The mapping to the Misconception Catalogue was done by a single researcher. These mappings are presented in Section 4.1, Tables 1 to 6.

3.3 Solution Development 2 - Differentiated Assessments

The purpose of this step was to modify a subset of the assessments that contain both prerequisites and course topics to make them differentiated, so that they can be used to diagnose whether a learner is struggling with prerequisites, course topics, or both. From the modified assessments, we also distilled a number of patterns and principles that can be applied to other assessments to make them differentiated.

When modifying questions, two researchers started by reviewing the previously coded prerequisite skills for the assessment and decided which of them to assess explicitly. We opted not to assess all prerequisites explicitly and individually in order to not increase the size of the question too much. This decision was based on two things: (a) the researchers' pedagogical content knowledge about student difficulties for the assessments's course topics, and (b) previous experience with the assessment in particular in their course and students' learning difficulties. This means that the modified exercises will only be able to indicate that *some* prerequisite is missing, or perhaps that one of a few well-known prerequisites are missing. Previous experience was available for all assessments except one (M.1) and the questions in the BDSI as none of the working group members had used them in their courses.

This phase was conducted in a similar manner to the previous one. First, two researchers set out to modify one question. This resulted in a number of patterns, only some of which were used, that were later generalized into the patterns and principles in the next phase. After this, other pairs of researchers modified other questions in the same way with the help of the previously identified patterns, thus validating and refining the patterns from the previous iterations. After a number of iterations of this process, modifying a number of questions, we arrived at the modified questions in Appendix M, some of which are presented in Sections 6.2 to 6.4.

3.4 Reflection to Produce Principles - PAPRIDA

In this phase, we examined the patterns and our experiences using them to create differentiated assessment (above), and generalized them into a number of patterns and principles for making assessments more differentiated, called PAPRIDA (PAtterns and PRinciples for Differentiated Assessment). The start of this phase essentially took place alongside the previous phase in the form of collecting patterns and associated experiences while modifying questions. The first researchers proposed a number of potential patterns after modifying the first questions. These were then explored by other researchers in the context of other questions, where they were refined, and new patterns were suggested. When all questions were modified, these patterns and the experiences related to them were collected and generalized to produce the list presented in Section 6.1. As these patterns and principles were refined throughout the assessment modification phase, the constant re-visiting and application of them serves as an initial validation of the patterns and principles. Note, however, that while PAPRIDA contains a set of useful patterns and principles for making questions differentiated, it is most likely not complete as it is rooted only on the structure of the initial questions collected.

4 RESULTS: PREREQUISITE SKILLS IN ADVANCED CS QUESTIONS

To answer RQ1: "What prerequisite skills do advanced CS questions depend on?", we analyzed a number of existing assessments as described in Section 3. Our main contribution related to RQ1 is the codebook produced during our empirical work of analyzing the advanced questions, which is presented below. We also present our coding of each of the individual questions. These two pieces of information contribute to answering RQ1. The codebook represents a set of possible prerequisites that should be considered as possible candidates for prerequisite skills in assessments. The coding of the questions contributes by giving concrete examples of the issue we aim to address in this paper, and which prerequisite skills they depend on.

4.1 The Codebook

We divided the codebook into the following six groups for ease of navigation:

- Basic Notional Machine
- Loops
- Values and Types
- Functions
- Objects
- High Level Skills

The codebook is presented in Tables 1 to 6 below. These tables also contain the mapping between our codes and the ACM 2013 Curriculum Guidelines, the Core Concepts and the Misconception Catalogue including an example misconception of each category. These mappings might aid the decision of which categories to include when modifying questions. For example, based on where they appear in the ACM Curriculum Guidelines, the estimated difficulty in the Core Concepts or by misconceptions that are known to be common in the Misconception Catalogue. By examining the mappings, we can see that all of our codes correspond to something in at least one of these three mappings.

Most of our codes correspond to one or more Knowledge Units in the ACM Curriculum Guidline. There are, however, exceptions such as *object scoping and data* which does not map to a Knowledge Unit. This does not mean that the ACM Curriculum Guidelines lack these particular topics, but rather that the Knowledge Units are at such a high level that no particular Knowledge Unit corresponds well to these codes. Another similar discrepancy is illustrated by the *operators* code, which maps to the *SDF/Fundamental Programming Concepts/Expressions and assignments*, which is too general to express this skill, and thus appear for multiple codes, such as *simple statements* and *assignments*. The same thing is true for the codes *return* and *return values*, which both map to the *SDF/Fundamental Programming Concepts/Functions and parameter passing* Knowledge Unit. From the mapping we can also see that the Core Concepts and the Misconception Catalogue lack some skills that are covered by our codes. This could be attributed to the fact that prerequisite skills vary depending on the context. For example, our code *API usage* represents a skill that is not necessarily an introductory skill, but still was a prerequisite skill for some of the analyzed assessments. However, some other skills were missing from these two, for example *return value*, which is not represented in either the Core Concepts or in the Misconception Catalogue.

4.2 Coding of Assessments for Course Topics

Tables 7 and 8 contain our coding of our initial 11 assessments (Table 7) and the 13 questions in the BDSI (Table 8). All of these questions are available in Appendices M and O.

From our coding of the questions, we can see that there are a number of questions that differ in their relation to prerequisite skills. We found three broad types of questions:

The first type is assessments that mostly focus on prerequisite skills. Not surprisingly, a typical example of such assessments are pre-exams that are given to students in the beginning of a course in order to assess prerequisite skills. These typically cover skills that are almost entirely captured by the skills presented in Section 4.1. See Appendix O.2 for an example of this kind of question.

Another type of questions that almost exclusively assesses course topics, meaning that almost none of the skills assessed by the question appear in Section 4.1. Many of the questions were found in the BDSI, and asked the student to reason about a data structure in higher-level terms. For example, see question M.6 in the Appendix. Other similar questions were found in a course on algorithm design, where the student was asked to reduce problems into suitable standard algorithms (Appendix O.4). This type of questions has an empty column for prerequisites, as can be seen in column M.6 of Table 7. It is worth pointing out that this does not mean that these questions do not have prerequisites; it only means that these prerequisites are not related to basic programming skills, but perhaps aimed more towards skills in algorithmic thinking, analytical thinking, logical reasoning, abstraction, or mathematics. They may also have prerequisites to actually learn the material, but the question itself does not require applying those prerequisite skills directly; for example, a question about the behavior of a data structure may not directly refer to code describing that behavior.

The final type of questions assess both prerequisite skills and course topics to a high degree. This is perhaps the type of questions that are most interesting to examine in the context of this research, as these typically have the property that if a student fails to answer a question, it is often difficult to attribute the failure either to the course topic or to a prerequisite skill. This does not, however, mean that such questions are undesirable. On the contrary, most questions of this type contained many prerequisites because they require the student to show that they are able to integrate the course topics into their prerequisite knowledge, and use it to solve some kind of real problems. Most of the questions we coded are of this type; for example see question M.5 in the Appendix with our coding of its prerequisites in Table 7.

5 RESULTS: IS THE BDSI ABLE TO DIAGNOSE PREREQUISITES?

To answer RQ2: "To what extent can an existing concept inventory for data structures with a validity argument – the BDSI – also diagnose difficulties with prerequisite skills?", we also examined all distractors for each question in the BDSI. For each distractor, we coded which prerequisites and course topics a student might have difficulties with when picking that distractor over the correct answer. From this analysis, we found that all 13 questions in the BDSI fall into five broad groups regarding how prerequisites and course topics interact:

Group 1 – B.4, B.9, B.13: These questions almost exclusively assess course topics (some of them assess meta-tracing knowledge as well). As such, they do not indicate problems with prerequisites (which is good in this case). For example, question B.4 asks about the time complexity of certain operations of a linked list, which does not depend on any prerequisites.

Group 2 – B.2, B.11: These questions contain both prerequisites and course topics, but none of the distractors allow drawing conclusions regarding whether a learner has difficulties with prerequisites or course topics.

Group 3 – B.1, B.3, B.5, B.8, B.10: These questions contain both prerequisites and course topics, and some of the distractors indicate that prerequisite skills are the issue rather than course topics. Other distractors do not make this distinction.

Table 1: The Codebook: Basic Notional Machine

Basic Notional Machine			
Skill	**ACM Knowledge Unit**	**Core Concepts**	**Misconception Catalogue**
Simple Statements: A basic understanding of statements in a language.	SDF/Fundamental Programming Concepts/Expressions and assignments		
Operators: Skills related to operators cover both being able to use arithmetic and comparison operators. Examples of these include questions related to operator precedence and Boolean logic.	SDF/Fundamental Programming Concepts/Expressions and assignments	OP (Operator Precedence), BOOL (Boolean Logic)	
Assignments: Understand how assignments work (e.g., the direction of the assignment), and that there is a difference between assignment and "equality" as used in mathematics.	SDF/Fundamental Programming Concepts/Expressions and assignments	AS (Assignment Statements)	VarAssign: "Primitive assignment works in opposite direction."
Basic input and output: These skills are related to being able to output messages and the value of variables to standard output and read data from standard input.	SDF/Fundamental Programming Concepts/Simple I/O		
Tracing: Being able to follow the step-by-step instructions in a program by utilizing knowledge of the notional machine while keeping track of the relevant state of the computation (e.g., variables, types).	SF/Computational Paradigms, application-level sequential processing	CF (Control Flow)	Ctrl: "Difficulties in understanding the sequentiality of statements."
Debugging: Refers to basic debugging skills to localise the bug, and to identify mismatch between intended outcome and actual outcome.	SDF/Development Methods, debugging strategies	DEH (Debugging/Exception Handling)	
Conditionals: How if- and switch-statements behave, for example that only one branch in an if-else statement is taken.	SDF/Fundamental Programming Concepts/Conditional and iterative control structures	BOOL (Boolean Logic), COND (Conditionals)	Ctrl: "Code after `if` statement is not executed if the `then` clause is."

Table 2: The Codebook: Loops

Loops			
Skill	**ACM Knowledge Unit**	**Core Concepts**	**Misconception Catalogue**
Loop constructs: Basic knowledge of the different looping constructs (typically, for, while and do-while). For example, that the condition is only executed before each time the loop body is executed in a for loop, and not between each statement.	SDF/Fundamental Programming Concepts/Conditional and iterative control structures	IT1 (Tracing execution of nested loops), IT2 (Understanding that loop variables can be used in expressions that occur in the body of a loop)	Ctrl: "`while` loops terminate as soon as condition changes to `false`."
Array iteration: Being able to utilize loops to iterate arrays, either using regular loops and indices, or any dedicated syntax for the task.	SDF/Fundamental Data Structures/Arrays	AR1 (Identifying and handling off by one errors when using in loop structures)	

Table 3: The Codebook: Values and Types

Values and Types			
Skill	**ACM Knowledge Unit**	**Core Concepts**	**Misconception Catalogue**
Types: Being able to associate a type to each variable, and trace the type information together with the value of the variable. This might involve finding and examining the type declaration in statically typed languages, or relying entirely on tracing in dynamically typed languages.	SDF/Fundamental Programming Concepts/Variables and primitive data types (e.g., numbers, characters, Booleans) and PL/Basic Type Systems/Association of types to variables, arguments, results and fields	TYP (Types)	VarAssign: "A variable is (merely) a pairing of a name to a changeable value (with a type). It is not stored inside the computer." Misc: "A type is a set of constraints on values.."
Values and references: The ability to differentiate between values and references (or pointers) to values, and the differences between making copies of a value and a reference to a value.	SDF/Fundamental Data Structures/References and aliasing	MMR (Memory Model/-References/Pointers), PVR (Primitive and reference type variables)	Refs: "Even primitive values (in Java) are handled through references."
Indirection: The ability to identify whenever a reference (or pointer) is traversed in order to access the value being referred to. Depending on the language, this might happen explicitly (e.g., pointers in C), or implicitly (e.g., accessing attributes in Java).	SDF/Fundamental Data Structures/References and aliasing	MMR (Memory Model/-References/Pointers), AR2 (Understanding the difference between a reference to an array and an element of an array)	Refs: "Once a variable references an object, it will always reference that object."
Arrays: Declaring and indexing arrays, and what happens when the index is out of bounds.	SDF/Fundamental Data Structures/Arrays	AR2 (Understanding the difference between a reference to an array and an element of an array), AR3 (Understanding the declaration of an array and manipulating an array)	Misc: "Confusion between an array and its cell."

Table 4: The Codebook: Functions

Functions			
Skill	**ACM Knowledge Unit**	**Core Concepts**	**Misconception Catalogue**
Parameters: Being able to declare and use function parameters to pass data into a function.	SDF/Fundamental Programming Concepts/Functions and parameter passing	PA1 (Understanding the difference between call by reference and call by value semantics), PA2 (Understanding the difference between formal parameters and actual parameters)	Sorva's "Methods" Topic is more related to OO design issues than function/method calls. There is a separate Calls Topics that deals with these issues.
Return values: Being able to declare and use function return values (or output parameters) to pass data out of a function. "Void" return values are included here.	SDF/Fundamental Programming Concepts/Functions and parameter passing		Calls: "A function (always) changes its input variable to become the output."
Return: Being able to use "return" to stop executing a function and return to the caller.	SDF/Fundamental Programming Concepts/Functions and parameter passing		
Function scoping and data flow: Being able to understand the scoping of local variables in a function, that the local variables are not shared between different invocations of the same function, and where parameters and return values fit in the model.	SDF/Fundamental Programming Concepts/Functions and parameter passing	PA3 (Understanding the scope of parameters, using parameters in procedure design), SCO (Scope)	
Recursion: The ability to understand and utilize recursive function calls, how execution flows through recursive functions and how values from operations are processed and stored. This is essentially only a special case that requires a better understanding of the three previous skills: *parameters*, *returns* and *function scoping and data flow*. It is however useful to separate this skill from the others, as recursion in itself is often problematic.	SDF/Fundamental Programming Concepts/The concept of recursion	REC (Recursion)	Rec: a Topic of its own.

Table 5: The Codebook: Objects

Objects			
Skill	**ACM Knowledge Unit**	**Core Concepts**	**Misconception Catalogue**
Classes/records/ADT: Being able to declare classes or records. This means to have understood their role in providing a *template* for instantiating objects which took part in the computation. It should be clear that classes define new (user-defined) types and they are units of encapsulation and scope, but not, for example, a way of ordering method calls (the order of method declaration is not relevant even w.r.t. their forward use in most OO languages).	SDF/Fundamental Data Structures/Records and structs (heterogeneous aggregates), SDF/Algorithms and Design/fundamental design concepts and principles, encapsulation and information hiding, PL/Object Oriented Programming, Definition of classes, methods and constructors, PL/Basic Type Systems/Compound types built from other types	CO (Classes and objects)	OtherOOP: "An object is just a record."
Object/instance/variable: Being able to differentiate between a class and an object (i.e., an instance of the class, or an instance of a type), or different instances of the same class. This involves being able to distinguish which objects (i.e., instances of a class or other memory entities of a built-in type) are active at a given point of a computation, in particular which objects have different identity (and, possibly, state) although they have the same type. The difference between references equality and object equivalence, the understanding of the problems of deep copies of complex objects is another important OOP basic skill. See also "Values and Types".		CO (Classes and objects)	ObjClass: "Confusion between a class and its instance."
Object scoping and data: Understand the lifetime of members of objects in relation to the object as a whole and the program. Class creation strategies (syntactic and semantic details can be quite different among different languages) define the lifetime of instance variables and methods (collectively known as *members*). A student should be able to differentiate among static visibility rules (`public`/ `protected`/ `private`/package in Java) and the accessibility of a specific entity, since even a private member can be reached by holding a reference returned by a method.		SCDE (Scope Design, understanding difference in scope between fields and local variables, appropriately using visibility properties of fields and methods, encapsulation)	ObjState: "During a method call, an object attribute is duplicated as variable. The local variable is initialized from the updated by the method, then returned to object at"
Static: Understand the difference between static and non-static members of a class. Member objects can be shared among the instances of the same class[1]. The role of `static` (to use the Java lingo) members and their special initialization rules should be well understood.		STAM (Static variables and methods)	

Table 6: The Codebook: High Level Skills

High Level Skills			
Skill	**ACM Knowledge Unit**	**Core Concepts**	**Misconception Catalogue**
Coding style: Understand and utilize elements of the language to make it easier to understand and reason about the code. Involves for example, comments, naming, use of empty lines, indentation, etc.	SDF/Development Methods/Documentation and program style		
API usage: Being able to find and use functions in some library (e.g., the standard library of the language). This involves searching for relevant functions, and reading the documentation to understand the semantics.	SF/Cross-Layer Communications/Programming abstractions, interfaces, use of libraries		
Problem decomposition: Being able to decompose a larger problem into smaller pieces with known solutions. For example, figuring out that a particular problem can be expressed in terms of a graph and use an appropriate graph algorithm to solve the problem.	SDF/Algorithms and Design/Fundamental design concepts and principles, Abstraction, Program decomposition	DPS1 (Design and problem solving 1, understands and uses functional decomposition and modularization), APR (Abstractions/Pattern Recognition and Use)	
Reasoning about constraints: Being able to reason about what is known and what is not known about the specification (i.e., preconditions) and reason about their implications on a particular piece of code or a particular method of solving a problem. For example, the array is not sorted, so using binary search is not possible without sorting the array first. Therefore, a linear search is faster.		DPS2 (Design and problem solving 2, Ability to identify characteristics of a problem and formulate a solution design)	
Meta-tracing knowledge: Knowing you need to go through some algorithmic process step by step to check an answer, executing/keeping track of a representation of computation. Knowing when you need to use an external representation (and the representation is good enough). Knowing where your limits are. For example, if a problem says: "trace this code", it probably does not involve meta-tracing since the problem explicitly tells the student to trace the code.		SVS (Syntax vs. semantics, understanding the difference between a textual code segment and its overarching purpose and operation)	

Table 7: The results of our qualitative coding of prerequisites assessed in the questions from advanced courses shown in Appendix M (questions that were eventually modified) and Appendix O (questions that were only used to devise the coding).

Qualitative codes	**M.1**	**M.2**	**M.3**	**M.4**	**O.1**	**O.2**	**O.3**	**O.4**	**O.5**	**O.6**	**O.7**
Simple statements	x	x	x	x	x	x	x		x	x	x
Operators	x	x		x					x	x	x
Assignments	x	x	x	x	x	x	x		x	x	x
Basic input and output									x		x
Tracing	x	x		x		x	x		x	x	x
Debugging		x									
Conditionals	x						x		x	x	
Loop constructs	x	x			x					x	x
Array iteration	x	x							x		x
Types	x	x	x	x		x	x				
Values and references	x	x	x	x					x		
Indirection		x	x	x					x		
Arrays	x						x				x
Parameters	x	x	x	x		x	x		x	x	x
Return values	x	x	x			x	x			x	x
Return									x		x
Function scoping and data flow	x	x	x			x	x		x	x	
Recursion									x	x	
Classes/records/ADT	x	x	x	x		x	x		x		
Object/instance/variable			x			x	x		x		
Object scoping and data						x	x				
Static											
Coding style											
API usage						x	x				
Problem decomposition		x						x			x
Reasoning about constraints	x	x						x			x
Meta-tracing knowledge	x	x	x	x		x	x	x	x	x	x

Table 8: The results of our qualitative coding of prerequisites assessed in the questions in the BDSI. Note that B.6 and B.8 were eventually modified and are therefore available in the Appendix.

Qualitative codes	**B.1**	**B.2**	**B.3**	**B.4**	**B.5**	**B.6**	**B.7**	**B.8**	**B.9**	**B.10**	**B.11**	**B.12**	**B.13**
Simple statements	x												
Operators						x	x						
Assignments	x					x							
Basic input and output													
Tracing	x												
Debugging											x		
Conditionals							x						
Loop constructs						x							
Array iteration													
Types													
Values and references	x	x	x			x							
Indirection	x					x							
Arrays													
Parameters							x					x	
Return values							x					x	
Return							x						
Function scoping and data flow							x					x	
Recursion							x					x	
Classes/records/ADT	x												
Object/instance/variable	x	x	x										
Object scoping and data													
Static													
Coding style													
API usage													
Problem decomposition										x			
Reasoning about constraints													
Meta-tracing knowledge	x	x	x		x	x	x		x	x	x	x	

Group 4 – B.6, B.12: These questions rely heavily on tracing skills, of course paired with some course topics. Therefore, an incorrect answer here likely means that some prerequisite is weak, perhaps in addition to some course topics. The distractors do not allow pinpointing the issue, however.

Group 5 – B.7: This question relies heavily on tracing skills, and as such incorrect answers mean that some prerequisite is weak. The combination of selected distractors do allow narrowing down the set of prerequisites quite well.

Below, we present a few examples of some questions in the BDSI along with an explanation of what skills the different distractors could indicate difficulties with. We have selected examples that illustrate the situation in groups 2 and 3, as they are the most interesting to explore further. We also provide a detailed breakdown of the possible answers to question B.7. All of the question statements are summarized for brevity, but the associated code and all possible answers, except for the correct answer, are reproduced verbatim. We have also rearranged the order of the answers and thus changed their labels. This is to make it more difficult for potential future takers of the test to find and memorize the correct answers to the BDSI online. Furthermore, in our analysis, the correct answer is generally uninteresting, and its omission does not impact the presentation of our results.

5.1 BDSI: Group 2, Question B.2

This question asks the student to compare two implementations of singly linked lists. One with a reference to the head of the list, and one with a reference to both the head and the tail. For each of the operations described below, the student is asked to select which (zero or more) operations would have better execution time (i.e., faster worst-case time complexity) in an implementation with a tail reference compared to one without a tail reference.

(a) Add a given element to the beginning of the linked list.
(b) Remove the last element from the linked list.
(c) Return `True` if the linked list contains a given element.

In this case, all answers involve course topics (in this case, mainly what a tail reference is). Thus, none of the distractors are able to tell whether a student fails to understand some aspect of a tail reference, or if they fail to understand the implications of that aspect due to lacking prerequisite skills.

5.2 BDSI: Group 3, Question B.1

This question asks the student to complete the following implementation of the `addAtTail` function, which adds an element at the end of a singly linked list that maintains both a `head` and a `tail` reference:

```
DEFINE addAtEnd(e)
  IF tail == nil THEN
    head = tail = new MyListNode(e)
  ELSE
    // MISSING CODE
  ENDIF
ENDDEF
```

The student is then asked to select one of the following five answers:

(a)
```
temp = new MyListNode(e)
tail = temp
```
(b)
```
temp = new MyListNode(e)
tail.next = temp
```
(c)
```
tail.next = e
tail = e
```
(d)
```
temp = head
WHILE temp.next != nil DO
  temp = temp.next
ENDWHILE
temp.next = new MyListNode(e)
```

Each of these answers highlight difficulties in prerequisites and/or course topics as follows:

(a) A student picking this answer over the correct answer could have difficulties with *object/instance/variable* (e.g., a student who does not understand that there are difference instances of `MyListNode` will likely not see the point in linking them), *meta-level tracing* (in this case, all operations only using the tail reference would work, while others do not), or with linked lists (e.g., forgetting that all nodes need to be reachable from the head).
(b) A student picking this answer over the correct answer likely has difficulties with either *meta-tracing knowledge* (in this case, the implementation works for one insertion, but not for two), or with linked lists (e.g., not understanding the purpose of the tail reference).
(c) A student picking this answer over the correct answer likely has difficulties with either *types* or *object/instance/variable*. The solution is correct, except that both `tail` and `tail.next` are supposed to refer to a node rather than an element. This could either be due to the student not realizing this through reasoning about the types, or by not realizing that a new node instance needs to be created.
(d) This solution would be correct in a linked list without a tail reference. As such, a student picking this answer over the correct answer is likely able to trace and understand the code, while only having difficulties with linked lists (in particular, tail references).

5.3 BDSI: Group 3, Question B.3

This question asks the student to investigate whether an implementation of a `LinkedList` class is using a singly linked list (with only a head reference), or a doubly linked list (with both head and tail references). The implementation is not provided, nor accessible. As such, the only option remaining is to conduct a small experiment (i.e., execute some of the operations on the list) and draw conclusions from there. The student is asked to select which of the following experiments is the best option:

(a) Create two instances of the `LinkedList` and test the timing of the first against the timing of the second. `LinkedList` for all `List` methods. If the timings between the first and second instances are close for all methods, the unknown `LinkedList` implementation is a singly linked list, otherwise it is a doubly linked list.
(b) Author a singly linked list class of your own and test the timing of it against the timing of the unknown list for all

methods. If the timing between your singly linked list and the unknown list is exactly the same for all methods, the unknown list is a singly linked list, otherwise it is a doubly linked list.

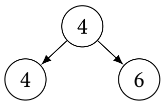
(c) Execute *n* `addAtEnd` operations followed by *n* `removeAtEnd` operations; if the `removeAtEnd` operations take much longer than the `addAtEnd` operations, we have a singly linked list, otherwise it is a doubly linked list.

(d) None of the above experiments would be able to answer this question. You would need to be able to examine the code to determine if the implementation uses a `previous` reference.

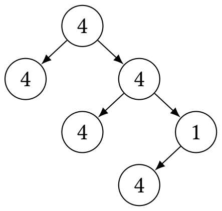
As is the case with question 2, most options involve both course topics (asymptotic notation and linked lists in this case) and prerequisites. As such, most distractors does not distinguish between the two. The distractor (a) is, however, interesting. It suggests that one can differentiate between a singly linked list and a doubly linked list by creating two instances of the unknown class, and doing the same sequence of operations to each of them in turn while measuring the time. For this to be true, the two instances of the class need to behave differently in some regard, which in turn implies some shared state between them. In particular, this would be true for a student who do not realize that different instances have a different set of member variables, and thus append all elements from the two instances to a single list. This reasoning implies that a student who picks the distractor (a) is likely to have difficulties with *object/instance/variable*.

5.4 BDSI: Group 5, Question B.7

This question asks the student to select all correct implementations of a function, `sum_leaves`, that computes the sum of all leaves in a binary tree (the exercise also contains a simple `TreeNode` class containing the variables `item`, `left` and `right`). The following two examples are provided to illustrate the expected behavior of the function:



`sum_leaves` should return 10



`sum_leaves` should return 12

The following four implementations are provided, and students are asked to select all that apply:

```
(a) DEFINE sum_leaves(node)
      value = 0
      IF node.left == nil AND node.right == nil THEN
        value = value + node.item
      ENDIF
      IF node.left != nil THEN
        sum_leaves(node.left)
      ENDIF
      IF node.right != nil THEN
        sum_leaves(node.right)
      ENDIF
      RETURN value
    ENDDEF
(b) DEFINE sum_leaves(node)
      IF node.left == nil AND code.right == nil THEN
        RETURN node.item
      ENDIF
      IF node.left != nil THEN
        RETURN node.item + sum_leaves(node.left)
      ENDIF
      IF node.right != nil THEN
        RETURN node.item + sum_leaves(node.right)
      ENDIF
    ENDDEF
(c) DEFINE sum_leaves(node)
      value = 0
      WHILE node != nil DO
        IF node.left == nil AND node.right == nil THEN
          value = value + node.item
        ENDIF
        IF node.left != nil THEN
          node = node.left
        ELSE IF node.right != nil THEN
          node = node.right
        ENDIF
      ENDWHILE
      RETURN value
    ENDDEF
(d) DEFINE sum_leaves(node)
      IF node == nil THEN
        RETURN 0
      ELSE IF node.left == nil AND node.right == nil THEN
        RETURN node.item
      ELSE
        RETURN sum_leaves(node.left)
             + sum_leaves(node.right)
      ENDIF
    ENDDEF
```

In this multiple choice question, it is interesting to explore all combinations of distractors selected by the student, as the combination can be used to better pinpoint which prerequisites (or course topics) that might be problematic:

(a)	*return value* or *recursion*
(b)	*return*
(c)	*meta-tracing knowledge*
(d)	-
(a,b)	*return* and *function scoping*
(a,c)	*recursion* and *meta-tracing knowledge*
(a,d)	*function scoping*
(b,c)	*return* and *loops*
(d,b)	*return*
(d,c)	*meta-tracing knowledge*
(a,b,c)	Likely *conditionals*
(a,b,d)	Likely *recursion*
(b,c,d)	You know *function scoping* and *return values*, but not *loops* and *return*

(a,b,c,d) Many areas: *meta-tracing knowledge*, *recursion* and *loops*. Likely only looks at the keywords and see if they appear.

From these examples, we can see that the distractors often do not distinguish between difficulties in prerequisites and course topics. In principle B.7 might be able to, as the many combinations of highlighted answers can be used to pinpoint the prerequisite difficulties quite well. However, students might also randomly guess or use other reasoning in practice, so empirical validity work with students would be needed to check the quality of these inferences in practice. This question does, however, not assess many course topics (in this case, only basic knowledge of trees) as it focuses on tree traversal.

6 RESULTS: DIFFERENTIATED ASSESSMENTS AND PAPRIDA

To answer RQ3: "What are examples of differentiated assessments and principles for designing differentiated assessments?", we utilized the results from RQ1 (Section 4) and modified a number of the analyzed questions to make them able to differentiate between difficulties with prerequisite skills and course topics. During these modifications, we also collected a set of patterns and principles to do these modifications, which we later generalized into PAPRIDA (PAtterns and PRinciples for Differentiated Assessment). In this section, we start by presenting the patterns and principles, followed by a detailed presentation of the analysis and modifications to one question from each of the three course topics: advanced object-oriented programming, data structures and concurrency.

Each of these questions will be examined in detail. For each question, we will present what course topics the question is designed to assess, along with some additional background. After that, we will describe the steps taken to modify the question: first the question is analyzed to find prerequisites. Then, a number of those are selected to be assessed explicitly. It is usually not wise to assess all prerequisites explicitly in a way that it is possible to distinguish between all of them as that would dramatically increase the size of the assessment considerably. Finally, we apply a number of patterns and principles from PAPRIDA to make the modifications to the question. Thus, these examples can be seen as "worked examples" of how to apply PAPRIDA to improve a question, either by making the question explicitly assess the prerequisites, or by introducing new prerequisites that are explicitly assessed to a question. Additional questions we modified are reported in Appendix M.

6.1 PAPRIDA: PAtterns and PRinciples for Differentiated Assessment

Below, we present PAPRIDA (PAtterns and PRinciples for Differentiated Assessment), which represents a set of patterns and principles that are helpful to explore in order to modify existing questions or to construct new ones.

Show your work: One relevant strategy instructors already use with questions to diagnose skills on the course topic is for learners to show all their work. This method does indeed reveal misconceptions in prerequisites, but is time consuming to grade, and learners may not actually show enough to diagnose their skills (especially if learners feel the allotted time is short). As such, depending on the context for the assessment, this strategy might not always be suitable. For example, if the goal is to add some small items to an already large assessment, it might be better to explore other strategies first.

Asking for details: One alternative to the previous approach is to add a small question that asks the student for some specific detail of the code in the question related to some prerequisite skills. This could, for example, be to ask the student to point to lines in the code that accesses a particular part in memory or asking for the type of a particular expression in a particular context. This is used in the concurrency exercises to assess whether students know when *Indirection* occurs. The benefit of this approach is that it does not increase the grading time by much, while still giving an indication of the prerequisite skills, but it might be difficult to find a small but precise enough such question. Another option is to insert a print statement that outputs something very specific and ask about that.

Altering terminology: One approach that might be used to assess new prerequisites to a question that previously assessed few of the prerequisites is to alter the terminology slightly. For example, instead of using high level terms, such as "accessing by index" one could use "`array.get(index)`" to require *Arrays* into the assessment. When used in the main question text, this does not necessarily make the question able to differentiate between prerequisite skills and course topics, but strategic use of this method, perhaps in a distractor, is useful for assessing this difference.

Introducing aliasing: One approach that was used to assess *Values and references* in the concurrency exercises was to break out parts of the code that modifies some variable into a new function with the data passed as a reference parameter. The formal and actual parameters should have different names so that the student has to make the connection between them explicitly. A print statement placed after the call to the new function can then be used to clearly see if a student understands which modifications are visible in the caller and which are not, and thus whether or not the student understands the difference between values and reference and their semantic when used as function parameters. This can thus also be used to assess *Function parameters* and *Indirection*.

Renaming variables: Another approach that was used in conjunction with the above one is to rename reference variables in different functions. For example, if multiple functions access the same data structure by reference, renaming the parameter so that it has different names in each function makes it impossible to rely on pattern-matching to arrive at the conclusion that they might refer to the same value, and thus introducing the *Values and references* skill. This can be properly assessed by adding a strategic print statement, or a short piece of code that executes two of the functions with the same data as parameter to check what student understands.

Adding another instance: A final approach that was used in the data structure and algorithms questions was to introduce multiple instances of a data structure in some part of the question. This requires students to be aware of the *Object* skill in order to be able to differentiate between the instances while tracing the implementation of some algorithm.

Adding distractors: In case of a multiple-choice question, having coded the prerequisites implicitly assessed in the question makes it possible to introduce additional distractors that explicitly address misconceptions related to the prerequisites. Note that this kind of distractors differ from those typically found in multiple choice questions. These distractors are concerned with misconceptions in prerequisites, and not misconceptions in the course topics. This approach is beneficial to pair with one of the others to introduce additional possibilities for creating relevant distractors.

Distractors with code: One interesting example we found in the BDSI was to have a number of distractors containing code, where the student need to select the pieces of code that are correct. This type of question opens up for checking many prerequisite misconceptions, as they allow each distractor to highlight a different set of them. Picking these sets with care allow all combinations of selections (assuming students may select more than one) to highlight one or a few of the misconceptions, making it possible to assess many possible misconceptions with a single question.

6.2 Advanced OOP: Inheritance and Polymorphism

In this section we will discuss our modifications to a question about inheritance and OOP. This assignment was designed to make students consider the problems that might arise when Liskov's substitution principle is not fully taken into account. In particular, if the pre-conditions for using a service provided by both a superclass and a subclass are stronger for the subclass, clients accessing subclass objects through references of superclass type might have constraints/expectations that will be not satisfied.

The question provides an implementation of the classes depicted in Figure 2 in Eiffel[2], and the code in Listing 1 that creates instances of the classes and calls the member function `eat` in various ways. The student is then asked whether different calls to `eat` is a compile-time or run-time error, and are then asked to add some constraints. The full question is presented in Appendix M.4.

To productively focus on the problem, the answering student should already master at least the following prerequisite skills (from our qualitative coding):

- Simple Statements
- Operators
- Assignments
- Tracing
- Types
- Values and references
- Indirection
- Parameters
- Classes/records/ADT

[2]The implementation language is Eiffel, a statically typed language in which method overriding can change formal parameters in a co-variant way, a choice that is not possible in Java or C++.

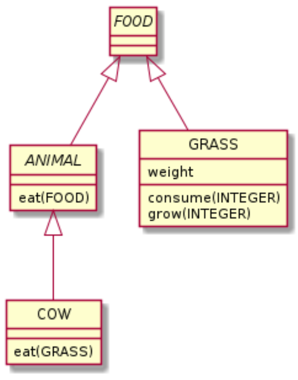


Figure 2: UML Class diagram for the polymorphism assessment

Listing 1: Part of the Eiffel code for assessment on Polymorphism (M.4)

```
class APPLICATION
create
  make

feature -- Main

  make
      -- Run application.
    local
      a: ANIMAL
      c: COW
      g: GRASS
      f: FOOD
    do
      create c
      create g
      a := c
      f := g -- focus on this
      print (a.out + " is going to eat: " + f.out + "%N")
      a.eat (f)
    end

end
```

- Meta-tracing knowledge

In particular, a clear understanding of references, the impact of their types, the handling of method parameters, and the ability to trace the flow of the computation are key for solving the exercise. Thus, it could be useful to add a couple of specific questions able to make evident the weaknesses or misconceptions in these areas. Tracing questions can be exploited in order to check that the acquaintance with the basic OOP notional machine is solid enough to support the OOP concepts. To this end, the assessment needs some updates. The `out` methods (see Appendix M.4 for the full source code) can be used to query and print the dynamic type of an object, therefore they can be leveraged on in tracing questions. Some useful modifications are (see Listing 2):

(1) Add a new concrete `FOOD` class (e.g., `PLANKTON`) and create an object `p` from this class (*Adding another instance*, a class in this case, from Section 6.1)

Listing 2: The modified Eiffel code for the modified assessment on Polymorphism (M.4). Additions are highlighted.

```
 1 feature -- Main
 2
 3   make
 4       -- Run application.
 5     local
 6       a: ANIMAL
 7       c: COW
 8       g: GRASS
 9       f: FOOD
10       p: PLANKTON
11     do
12       create c
13       create g
14       create p
15       g.grow (5)
16       a := c
17
18       -- log_food2(f) -- log_food(f) not legal
19       f := g
20
21       log_food(p)
22       log_food(g)
23       log_food(f)
24
25       print (a.out + " is going to eat: " + f.out + "%N")
26       a.eat (f)
27       print ("Finished!%N")
28     end
29
30     log_food(x: FOOD)
31     do
32       print ("The food x is: " + x.out + "%N")
33     end
34
35     log_food2(x: detachable FOOD)
36     do
37       if attached x then
38         print ("The food x is: " + x.out + "%N")
39       else
40         print("No x%N")
41
42       end
43     end
44 end
```

(2) Add a method `log_food` with a parameter `x` of type `FOOD`, the method just prints the dynamic type of the actual parameter bound to `x` (*Asking for details* from Section 6.1)
(3) Add a question about the output of `log_food(p)`, `log_food(g)`, `log_food(f)`, where `g` is a reference to a `GRASS` object and `f` is a reference to a `GRASS` object of `FOOD` static type. Note that it would not be legal to call `log_food` with `f` as an actual parameter before assigning it to a concrete object. In order to do this, the type of the parameter must be marked as `detachable`[3]. This observation can be used to assess the students' understanding of nullable references by asking why the call to `log_food2` at line 18 is legal while `log_food` is not. (*Asking for details* from Section 6.1)

6.3 Data Structures

In this subsection, we describe our modifications to a data structure exam problem. The problem is available in Appendix M.1, but we provide a summary here. The question presents students with the code in Listing 3 and are asked to:

(a) determine whether it implements a stack, a queue, a priority queue or a union find data structure
(b) implement a suitable `size` method (from four options)
(c) determine which out of four possible invariants are upheld by the data structure
(d) trace the behavior of a sequence of insertions and removals
(e) reason about the number of array accesses the `remove` method performs in the worst case
(f) reason about the number of array accesses the most expensive public operation perform in the worst case
(g) reason about a generic sequence of operations
(h) reason about memory consumption of the data structure

Listing 3: Code from the data structure question (M.1)

```
public class Y<Key extends Comparable<Key>>
{
    private Key[] A = (Key[]) new Comparable[1];
    private int lo, hi, N;
    public void insert(Key in)
    {
        A[hi] = in;
        hi = hi + 1;
        if (hi == A.length) hi = 0;
        N = N + 1;
        if (N == A.length) rebuild();
    }
    public Key remove() // assumes Y is not empty
    {
        Key out = A[lo];
        A[lo] = null;
        lo = lo + 1;
        if (lo == A.length) lo = 0;
        N = N - 1;
        return out;
    }
    private void rebuild()
    {
        Key[] tmp =
            (Key[]) new Comparable[2*A.length];
        for (int i = 0; i < N; i++ )
            tmp[i] = A[(i + lo) % A.length];
        A = tmp;
        lo = 0;
        hi = N;
    }
}
```

The code included in the question implements a queue with a circular array and two integers: `lo` is the index of the first element in the queue and `hi` is the index just after the last element in the queue. The variable `N` represents the number of element in the queue. The array is rebuilt with doubled size whenever the insertion of an element exhausts the capacity of the array.

The question assesses the following topics on data structures:

- to distinguish among different data structures;
- to understand code implementing a data structures;

[3]In Eiffel types are by default "attached", meaning that they do not permit void (null) values: to support null references, a type must be declared as detachable.

- to know worst case and amortized complexity notion;
- to analyse the complexity of an algorithm expressed by a piece of code

From our analysis of the question, we can also see that it requires the student to understand the following prerequisite concepts, even though none of them is assessed explicitly:

- Simple statements
- Operators
- Assignments
- Tracing
- Conditionals
- Loop constructs
- Array iteration
- Types
- Values and references
- Arrays
- Parameters
- Return values
- Function scoping and data flow
- Classes/records/ADT
- Reasoning about constraints
- Meta-tracing knowledge

By further examining the different items, we can see that items (a) to (d) require close inspection of the code to figure out how the data structure works, which in turn require skills related to code comprehension. For example, lines 8 and 17 can be used to rule out the possibility that the data structure is a stack for item (a), lines 10 and 19 can be used to identify that N is indeed the number of elements in the stack for item (b), and understanding the circularity and the rebuilding policy are essential for answering (c) and (d). Items (e) and onwards are then more focused on course topics (in this case, mostly asymptotic analysis) while relying on prerequisites to a lesser extent.

Throughout the exercise, the most critical prerequisites are: operators (modulus in this case), conditionals, arrays and array iteration. As these are not assessed explicitly anywhere in the question, and are deemed critical, we focused our modifications on making these prerequisites explicit. Our changes to accomplish this are outlined below. The full modified question is presented in Appendix M.1.

- A new items was added at the end of the tracing task in item (c), that asks how many times the rebuild method is called. This assesses the prerequisite knowledge on conditionals and hence ascertains that the building policy has been understood (*Asking for details* from Section 6.1).
- Another concrete tracing task was added to assess knowledge on operators (modulus), arrays (indexing and storage), and array iteration (*Asking for details* from Section 6.1).
- An item was added that proposes a specific input situation and asks to establish if it can occur after a sequence of insert and remove call. This requires the students to reason about the pre- and post-conditions and the invariants of the code (*Asking for details* from Section 6.1).
- We finally add an item to assess knowledge about values and references. Additionally, this exercise also assesses the ability to differentiate between an object/ADT and its instances, which was deemed an important and related prerequisite skill, even though it was not strictly necessary to answer the original question. More precisely, students are asked to establish the values of variable a and b after executing the following piece of code (*Adding another instance*/*Introducing aliasing* from Section 6.1):

```
Y y = new Y();
Y z = new Z();
Y w = z;
w.insert(3);
z.insert(1);
y.insert(2);
int a = z.remove();
int b = y.remove();
```

6.4 Concurrency/Synchronization

The analyzed questions assessing concurrency were designed to explicitly assess *threads*, *busy-wait*, and the student's ability to use suitable synchronization primitives (out of *semaphores*, *locks* and *condition variables*) to solve synchronization issues. This can be quite clearly seen from the original question in Appendix M.2. The question presents students with the code in Listing 4, which implements a buffer containing strings, and asks students to identify and solve occurences of *busy-wait*, and then to identify and solve any remaining synchronization issues. By analyzing the question, we can see that it also requires the student to understand the following prerequisite concepts, even though they are not explicitly assessed:

- Simple Statements
- Operators
- Assignments
- Tracing
- Debugging
- Loop constructs
- Array iteration
- Types
- Values and references
- Indirection
- Parameters
- Return values
- Function scoping and data flow
- Classes/records/ADT
- Problem decomposition
- Reasoning about constraints
- Meta-tracing knowledge

By examining the study by Strömbäck et al. [86], which studied students' performance on this particular question, we can see that the authors observed that it was not always clear if particular categories of incorrect answers were due to students not understanding concurrency, or had some misconceptions regarding different instances and/or pointers. The authors also noted that some students attempted to synchronize local variables in functions, suggesting that students either do not understand what is relevant to synchronize, or that students do not understand function scoping. Because of this, we want to explicitly assess the skills *Objects*, *Values and references*, *Indirection* and *Function scoping and data flow* in order to be able to properly differentiate difficulties in prerequisites from difficulties with the course topics.

Listing 4: Code for the concurrency exercise (M.2)

```c
struct idea_buffer {
  // All ideas in the buffer. Empty elements are
  // set to NULL.
  const char *ideas[BUFFER_SIZE];
  // Number of ideas in the buffer.
  int count;
};
// Add a new idea to an empty location in the
// buffer. Returns 'false' if the buffer is full.
bool idea_add(struct idea_buffer *buffer,
              const char *idea) {
  int found = BUFFER_SIZE;
  for (int i = 0; i < BUFFER_SIZE; i++) {
    if (buffer->ideas[i] == NULL) {
      found = i;
      break;
    }
  }
  if (found >= BUFFER_SIZE)
    return false;
  buffer->ideas[found] = idea;
  buffer->count++;
  return true;
}
// Get and remove a random element from the
// buffer. If no elements are present, the
// function waits for an element to be added.
const char *idea_get(struct idea_buffer *buffer) {
  while (buffer->count == 0)
    ;
  buffer->count--;
  int pos = rand() % BUFFER_SIZE;
  while (buffer->ideas[pos] == NULL) {
    pos = (pos + 1) % BUFFER_SIZE;
  }
  const char *result = buffer->ideas[pos];
  buffer->ideas[pos] = NULL;
  return result;
}
```

The modified exercise is presented in Appendix M.2. The modifications address the prerequisite skills through the following modifications:

- The name of the pointer variables referring to the shared `idea_buffer` were altered in order to make it impossible to rely on pattern-matching to realize that the variables inside the struct may refer to the same variable, thus requiring the student to understand how pointers work in order to be able to identify shared data. This is explicitly assessed by adding the following question (*Renaming variables* from Section 6.1):
  After executing the following code, what is the value of the variable `res`?

```c
struct idea_buffer x;
idea_init(&x);
idea_add(&x, "a");
int res = x.count;
```

- In order to assess the *object* skill explicitly, we add the following question that requires the student to be able to differentiate between different instances of the same struct (*Adding another instance* from Section 6.1):
  What is the expected behavior when executing the last line in the code below?

```c
struct idea_buffer a, b;
idea_init(&a);
idea_init(&b);
idea_add(&a, "a");
idea_get(&b); // <-- here?
```

- In order to assess the *indirection* skill explicitly, a question asking the student to mark all locations where data inside a `idea_buffer` is accessed (*Asking for details* from Section 6.1).
- Finally, in order to assess the *function scoping and data flow* category, students are asked which variables are not possibly shared between threads. If students fail to mark any of the variables, it indicates that the student might incorrectly believe that local variables are shared between threads, and thus that the student might not understand function scoping properly (*Asking for details* from Section 6.1).

By adding these parts to the question, the student will get some help examining the code for things relevant to the synchronization tasks, but more importantly, the instructor will be able to examine the answers to these parts and assess whether any mistakes in other parts of the question are due to concurrency or not.

7 LIMITATIONS

As our research is an initial investigation into nature of prerequisite skills in advanced courses and possibility of differentiated assessments to explicitly assess prerequisites, there are a number of limitations.

Small, non-representative question sample. We did not attempt to systematically sample assessment questions from instructors, nor did we attempt to make generalizable claims about the frequency of prerequisite skills appearing in course topic assessment questions. The sample size of questions used to develop the codes was small and not representative of all questions, although this is not abnormal in qualitative analysis. We included the BDSI for its validity argument and careful question design, to get qualitative insights on prerequisite dependencies for questions made with existing concept inventory design methods. We also included questions used previously by the working group members, where the members previously experienced that students had trouble with due to lacking prerequisite skill. Due to the working group's interest in tracing as a tool for assessing these skills, we tended to pick examples that involved code comprehension and/or tracing problems. This small sample size and possible bias is always present in our kind of qualitative analysis, which instead merely aims to show existence of prerequisite issues and provide examples of how they appear in questions. Evidence of existence is not evidence for their frequency or magnitude in different contexts.

Qualitative analysis by experts. The qualitative coding of prerequisites required for questions, which is not an empirical study of students' behavior, relied on our understanding of students' mental models and informally remembered behavior of students on questions. Our understanding may not be consistent with actual learner behavior if we empirically studied their behavior. We did follow good inductive coding methods where two researchers independently coded each question first, then discussed disagreements, revised our codes for clarity, and finally iterated to consensus, but ultimately it is still an expert analysis. Our consensus process only applied to our coding of our data, the reliability of the codebook on other data is not known and should be measured in future work (i.e. how well others can use the codebook to consistently code new questions).

Non-exhaustive analysis of some prerequisites: basic and tracing-related skills. The codes are not an exhaustive or complete listing of all prerequisite skills required for our questions (for example, we did not code for needing to understand English in questions, mathematical skills, etc.). Even a computing-only complete listing of possible prerequisites would be very difficult as CS is taught differently at different institution, and as such any list of prerequisites smaller than most of the entirety of CS will inevitably lack some possible prerequisites in some context. Since we have examined questions from different advanced topics (concurrency, data structures & algorithms, and advanced OOP), our inductive coding has some coverage of the major prerequisite topics brought up, at least regarding imperative and object-oriented languages.

Furthermore, due to the small and possibly biased set of initial questions, the list of prerequisite skills may not be complete. This is not a major issue in the context of this report, since our goal is to show that prerequisite skills are implicitly assessed in assessments for later courses and how to improve assessments. Additionally, even though the list might be incomplete, it is still useful to highlight a number of common prerequisites and address those.

Our modified assessments and lack of empirical evaluation with students. Finally and most importantly, we did not empirically evaluate or make a validity argument for the questions modified in this paper (listed in full in Appendix M) with actual students. Instead, we once again relied on our understanding of students' mental models during this phase. Even though we personally believe that the modifications do indeed improve the questions to more explicitly assess prerequisites of the questions, that is only our judgement of face validity. Additional empirical validity work with actual students needs to be done to ensure the modifications have their desired properties, and is described in our future work. For now, our example modified questions should be seen as worked examples of how to apply the PAPRIDA suggested in this paper — more validity work is required for evaluating them.

There were also many potential ways to modify the questions, including which prerequisites to target, but we only explored a few examples, which means there may be many other principles and patterns that we did not discover.

8 DISCUSSION

In this section, we will examine our results and discuss in what context they are useful, and what new ideas they bring to the table. First, we summarize our results in general here, then we discuss our results for each research question more deeply. We then discuss how making *differentiated assessments* raises questions for theories of computing knowledge and how we might refine such theories by analyzing assessments. Lastly, we discuss other areas of future work, including applying our paper's approach to other course topics. In addition, we recommend empirical validity studies for our initial differentiated assessments.

8.1 Summary of Results by Research Question

The hypothesis that motivated our work is that assessments for advanced courses may not differentiate between lacking prerequisite skills and course topics. Our results provide support for our hypothesis, implying that this issue is present to some extent and worthy of future research.

For RQ1 "What prerequisite skills do advanced CS questions depend on?", we found variation among advanced assessment questions in their required prerequisite skills, often requiring many such skills, and sometimes none.

For RQ2 "To what extent can an existing concept inventory for data structures with a validity argument – the BDSI [66] – also diagnose difficulties with prerequisite skills?", we found even for high quality questions, many unable to precisely diagnose prerequisite skill issues from incorrect answers. Of the BDSI questions, ten required prerequisites, and one of those ten was diagnostic (see Section 5.4), five had some distractors indicating a small set of prerequisites and/or advanced skills, and four could not diagnose (Groups 2 and 4 in Section 5).

Thus, for RQ3, our goal was to investigate the feasibility of designing assessments that are able to differentiate between the two, ideally with no or minimal increase in the time required neither for students to do the assessments, nor for teachers grading the assessments. We then made six modified assessment questions, designed to better differentiate (according to our judgement as instructors), and distilled initial patterns and principles for differentiated assessments (PAPRIDA), including computing-specific patterns like "introducing aliasing".

8.2 Research Question 1

Most questions we analyzed required some prerequisite knowledge. For example, question M.1's course topic is data structures, but learners might get parts of it incorrect due to not being able to trace the loop inside the `rebuild` function. In this case, it might be due to a lacking understanding of loops, or a lacking understanding of operators (modulo in this case), which could make the student believe that the function does something more advanced than resizing the data structure's array.

Some questions did not require any of the prerequisites we analyzed. These interesting questions were more conceptual, for example, some high level data structures questions on the BDSI (see, for example, question B.4 in Table 8). These questions can be useful for finding and correcting gaps in conceptual knowledge. Future work should find ways to make such focused and specific questions, contributing general patterns and ones specific to computing education specific.

While it can be good to have such questions, assessments that only cover the course topics are not necessarily better. First, students might be able to answer such questions by rote memorization, seemingly knowing part of the course topic without knowing prerequisite skills; for example, for a question asking for the big-O runtime for an algorithm, a learner may answer correctly without understanding how to derive it from the algorithm or what it really means. Second, if we mostly use questions solely on the course topics, we might teach and assess course topics in an isolated way, without practicing the prerequisites you need to use them. Without practice learners can forget or become weaker in those prerequisites over time.

8.3 Research Question 2

The results of our qualitative analysis of the BDSI distractors showed few questions could diagnose prerequisite issues very specifically. Given our results, concept inventories for course topics should not be presumed useful for diagnosing prerequisite issues. This is not surprising, since this is not a design goal of the BDSI or concept inventories in general; nor do they usually desire to make validity arguments that individual questions have validity. This connects to related work in educational assessment we described in Section 2.4; one should not assume concept inventories and other assessments have desirable properties, such as diagnosing learner's misconceptions [28, 73].

8.4 Research Question 3

The main practical contribution of this work is a toolbox of patterns (listed in Section 6.1) for augmenting assessments to be more differentiated. We have found a few initial computing specific technical tricks, like "Introducing aliasing" or "Renaming variables", that have the underlying goal of testing which chunks of prerequisite knowledge the solver is able to use in a new, more advanced, setting. These strategies are particularly useful when a large number of students are expected to take the modified assessment, as it may require less manual grading compared to, for example, "Show your work" questions, while still being able to diagnose some prerequisites issues. For example, as in the concurrency question in Section 6.4, one can carefully re-structure code in an assessment and/or add question parts that highlight the relevant prerequisites. This restructuring can enable good feedback on prerequisites without increasing the workload for the student considerably, and also may be easy to grade automatically and thus give immediate feedback to students. At the same time, our PAPRIDA recognize patterns teachers have used for millennia to get students to reveal their problem solving process, like "Show your work" and "Asking for details".

The idea of augmenting assessments to be more differentiated is different from a pre-exam at the start of an advanced class: in fact, students have probably *passed* such an exam, but this is in many cases not enough to guarantee that prerequisite skills are at the level required to focus properly on the new ones. Differentiated assessments can be more narrowly focused on what is actually instrumental to course topics, and this might also help students in making sense of what they learn.

8.5 Implications for Research on Theories of Computing Knowledge

Our research raises questions for theories of computing knowledge. Theories of computing knowledge include, for example, "theories of what it means to know a programming language, what it means to know how to program, what it means to be an expert software engineer, what it means to have computer science literacy, and numerous other unanswered and yet foundational questions that are specific to computing education" [56]. Our work raises questions such as: For every advanced topic in computing, can a person know parts of it without knowing the prerequisites? Is it desirable to make questions that purely assess an advanced topic? For what topics is it possible to do that and why? How can we theoretically specify and separate shallow, fragile ways learners may know these skills from more fluent and transferable knowledge?

Perhaps making assessment questions is a useful way of operationalizing theories of computing knowledge, as a kind of design-based research program [3, 7]. For example, researchers might try to make more specific questions (as a 2018 ITiCSE WG did for programming fundamentals [47]), try the assessments with actual learners, then iterate on their theory of computing knowledge, by perhaps questioning skills they can not seem to assess, or adding skills inspired by an assessment they made.

As a concrete example within our work, one prerequisite knowledge code arose that we called "meta-tracing". It represents a set of self-regulated problem solving behaviors and competency for applying representations of computation appropriate for problem solving, for example, "Knowing you need to go through the steps to check your answer, step by step." This qualitative code might be further developed in later research - separated into components, empirically checked to see if learners seem to develop a general skill like this or if it just specific to learning particular problem types.

Our community might also want to analyze existing assessments made by teachers to develop theories of computing knowledge. This would draw on teachers' sense of what knowledge and skills are important to assess, which are expressed in question designs.

8.6 Implications for Instructors and Teaching

Assessments and assignments that differentiate at least some prerequisite skills, can help instructors and students. For example, an early set of differentiated questions can be "disguised" as quizzes on the first few lectures, which gives most of the benefits of a pre-exam, while also giving the students the opportunity to practice on course topics. This is opposed to taking yet another pre-exam containing only prerequisites. In addition to keeping students motivated to take the test, this also utilizes the instructors' valuable time with students more efficiently. Assessing prerequisites explicitly in midterms and final exams is also beneficial to catch cases where a student is not proficient enough with the prerequisites to be able to focus on the course topics. Instructors can also find common weaknesses in the specific prerequisite knowledge, may realize why their tests are too hard and try to make learning steps more explicit and give more feedback to students.

Differentiated questions seem especially promising for improving equity. The more students come in with varied backgrounds, especially in more advanced degree programs, the more beneficial they are for both identifying students falling behind and helping them with targeted feedback. The questions do not need to cover everything. If the assessment is able to tell that *some* prerequisite is lacking, the student can be instructed to take a more extensive test. This could check all prerequisites in depth, and the student can be directed to further practice the relevant subjects.

In making differentiated assessments, it can be infeasible to assess all the prerequisites diagnostically. Instructors making differentiated assessments should prioritize, by asking TAs to gather common difficulties or drawing on their experience in class and office hours when choosing which prerequisites to prioritize.

Instructors might benefit by following our paper's process of analyzing and modifying their existing questions to make them more differentiated. Instructors might surface and reflect upon the effectiveness of their assessments and their inclusion of prerequisite skills, intended and unintended. Using this information, the teacher might improve their teaching, via a more informed decision of what prerequisites might be beneficial to assess explicitly and address, or perhaps which could be excluded from the assessment. Instructors teaching an advanced class could go through that process together within or across institutions.

8.7 Future Work

In this section, we outline empirical validity studies for our theoretical findings presented in this report. We also discuss how our work leads to other interesting research topics for future investigation, which our community can pursue, such as making differentiated assessments for other advanced topics and prerequisites, and new patterns and principles for designing differentiated assessments.

8.7.1 Evaluating questions empirically for validity properties, especially for formative use. As mentioned in our limitations (Section 7), the modified questions have not been empirically evaluated with students. A validity argument based on, for example, Kane's framework [30, 55] should be made using empirical studies, such as think-alouds and using the questions in actual learning environments.

To evaluate how well questions differentiate prerequisite skills and advanced skills, learners can take a differentiated assessment, then separate advanced topic and prerequisite assessments; the study can compare how well they match. The study may also interview learners or evaluate think-alouds to diagnose the learner's knowledge, then compare with the diagnosis of the differentiated assessment.

To evaluate the modified questions' validity for giving feedback, learners can take a differentiated assessment, then the instructor could give feedback based on the assessment's diagnosis, automated or manual. Another design could involve using differentiated assessments in a class, then comparing any improvement in learning outcomes and the equity of learning process, during or at the end of the course.

In particular, our modified BDSI questions don't have empirical validity work and shouldn't be used in place of the original versions (as with questions for any assessment with a validity argument). Empirical validity work needs to be done for any question modification in general, and adding a question to a test may require redoing validity studies for the entire test.

8.7.2 Creating differentiated assessment questions for each advanced computing topic and for different prerequisites. We made several example questions for a small part of three advanced topics, scoping prerequisites to focus on the syntactical and conceptual knowledge of basic programming constructs, and on program comprehension skills including code tracing. Future work can be done for many different choices of advanced topics, and many different choices of prerequisites. That work might also contribute new patterns and guidelines for differentiated assessments.

This future work should evaluate the generality of our PAPRIDA, the patterns and principles presented in Section 6.1. Our paper's design method can be applied to different questions for advanced courses, both inside and outside of the topics covered in this report. This would involve coding a question according to the codebook in Section 4.1, determine which prerequisites are relevant to assess, and applying the patterns in Section 6.1 to make them explicit, and ideally empirically evaluate the resulting questions. This work may also evaluate how well our PAPRIDA generalize, and contribute new PAPRIDA. There are likely computing education specific question patterns for each area of knowledge, just waiting to be discovered and distilled.

8.7.3 Analyzing curricular assumptions using prerequisite coding of assessment questions. The coding of prerequisite skills presented in this paper could be used to examine the pedagogical assumptions made in the curriculum (sequence of courses) by analyzing their assessments. To do this, one would code the prerequisite skills assessed by some course, and code the skills assessed in any previous courses. The codes for the examined course and the prerequisites can then be compared to find any skills assessed as prerequisites in the advanced course, but not assessed or taught in any of the previous courses. Such a discrepancy indicates that either the expectations of the latter course need to be lowered, or that some of the previous courses need to be expanded to include the missing prerequisites.

8.7.4 Extending our paper's assessment design method to also include qualitative coding of advanced course topics. In this paper we only code skills that were considered prerequisites to at least one of the advanced topics. It would be useful to extend our method to include some advanced skills as well, using the method presented in Section 3.2. This would extend the codebook to include codes for advanced skills. Such an extension would also allow exploring curricular assumptions from earlier advanced courses to later advanced courses (see Section 8.7.3).

8.7.5 Developmental stages for course topics. Another interesting extension to the framework is to introduce *developmental stages* for the course topics. These could be based on the coding of course topics as mentioned above, but also from other observations in the literature. For example, in a course on concurrency, we can conjecture that a student start by being able so solve simple, explicit synchronization goals, then progresses to be able to identify some shared data and synchronize it properly at a coarse level, and finally at a finer level. Given these developmental stages, one could then create a matrix with developmental stages on the horizontal axis, and a set of tracing weaknesses from Section 4.1 that are known to be problematic in the context (if not all of them). Then, for each cell in the matrix, one would write what a hypothetical student with the given tracing weakness at the given developmental stage would be able to do, and what the student would not be able to do. This information could then be used to select appropriate questions during a course, or to design questions that are able to diagnose at what developmental stage a particular student is at, and which, if any, tracing weaknesses the student have.

9 CONCLUSION

In this paper we presented and discussed a novel methodology for revising assessments of advanced course topics to be more diagnostic, by making assessment of prerequisite knowledge explicit. We qualitatively analyzed existing questions, including the BDSI, a cutting edge high quality data structures assessment, and found interesting issues and implicit assessment of prerequisite knowledge (see Sections 4 and 5). We developed PAPRIDA, initial patterns and principles for designing differentiated assessments, which may help better diagnose issues with prerequisite knowledge (see Section 6). We made six example modified assessment questions (see Sections 6.2 to 6.4 and Appendix M).

Achieving more *differentiated* assessments for advanced topics and prerequisites is a worthy research goal contributed by our paper. The key idea is to expand the typical scope we define for an assessment, expanding it to also diagnose student issues with prerequisites. There is much work that remains to be done. For example, our assessment design work requires future validity studies and empirical work with learners to evaluate our PAPRIDA and example questions. Ultimately, differentiated assessments should be evaluated by their usefulness to the learning process by directly evaluating learning gains from using them to give formative feedback to learners.

We chose to focus on tracing skills for prerequisites and data structures, advanced object-oriented proramming, and concurrency as our advanced course topics. We invite readers to follow the approach in this paper for other scopings of prerequisite and advanced course topics. Our approach was driven by reviewing prior skill classifications, then qualitative coding prerequisite knowledge in a sample of assessment questions, then revising assessments to make more explicit, diagnostic assessment of those prerequisites. The key idea is that through the coding process itself, researchers and teachers are prompted to critically examine assessments from the perspective of prerequisite knowledge, and may then be able to better identify such topics where they are implicitly assessed. Therefore we emphasise the importance of the qualitative coding process, to the overall assessment design process, as a part of ongoing key work in computing education research developing theories of computing knowledge and making corresponding assessments with validity arguments.

ACKNOWLEDGMENTS

We thank the working group chairs and reviewers for their constructive feedback and service during the difficult and busy times during the pandemic.

We thank the BDSI team for their blessing and feedback about how to reprint two of their questions. The BDSI concept inventory is available in their *Google group*[4].

We thank Andrea Verdan for doing think-alouds with some draft questions and the BDSI as part of coding those questions, and proof reading. We also thank Ibrahim Albluwi for formative review of prior work, even though he had to leave the working group early.

REFERENCES

[1] Wendy K Adams and Carl E Wieman. 2011. Development and validation of instruments to measure learning of expert-like thinking. *International Journal of Science Education* 33, 9 (2011), 1289–1312.

[2] Association for Computing Machinery. 2013. Computer Science Curricula 2013. https://www.acm.org/education/curricula-recommendations

[3] Sasha Barab and Kurt Squire. 2004. Design-Based Research: Putting a Stake in the Ground. *Journal of the Learning Sciences* 13, 1 (2004), 1–14. https://doi.org/10.1207/s15327809jls1301_1 arXiv:https://doi.org/10.1207/s15327809jls1301_1

[4] Ricardo Caceffo, Steve Wolfman, Kellogg S. Booth, and Rodolfo Azevedo. 2016. Developing a Computer Science Concept Inventory for Introductory Programming. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education (SIGCSE '16)*. Association for Computing Machinery, New York, NY, USA, 364–369. https://doi.org/10.1145/2839509.2844559

[5] Nicholas Carriero and David Gelernter. 1989. Linda in Context. *Commun. ACM* 32, 4 (April 1989), 444–458. https://doi.org/10.1145/63334.63337

[6] Tony Clear, J. Whalley, P. Robbins, A. Philpott, A. Eckerdal, and M. Laakso. 2011. Report on the final BRACElet workshop: Auckland University of Technology, September 2010. *Journal of Applied Computing and Information Technology* 15 (Jun 2011). Issue 1. http://aut.researchgateway.ac.nz/handle/10292/1514

[7] Allan Collins, Diana Joseph, and Katerine Bielaczyc. 2004. Design Research: Theoretical and Methodological Issues. *Journal of the Learning Sciences* 13, 1 (2004), 15–42. https://doi.org/10.1207/s15327809jls1301_2 arXiv:https://doi.org/10.1207/s15327809jls1301_2

[8] Malcolm Corney, Sue Fitzgerald, Brian Hanks, Raymond Lister, Renee McCauley, and Laurie Murphy. 2014. "Explain In Plain English" Questions Revisited: Data Structures Problems. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education (SIGCSE '14)*. Association for Computing Machinery, New York, NY, USA, 591–596. https://doi.org/10.1145/2538862.2538911

[9] L. J. Cronbach and P. Meehl. 1955. Construct validity in psychological tests. *Psychological bulletin* 52 4 (1955), 281–302.

[10] J. H. Cross II, T Dean Hendrix, and Larry A Barowski. 2002. Using the debugger as an integral part of teaching CS1. In *32nd Annual Frontiers in Education*, Vol. 2. IEEE, Stripes Publishing LLC, Chapaign, IL, USA, F1G–F1G. https://doi.org/10.1109/FIE.2002.1158137

[11] Dana Denick, Aidsa Santiago-Román, Ruth Streveler, and Natalie Barrett. 2012. Validating of the diagnostic capabilities of concept inventories: Preliminary evidence from the Concept Assessment Tool for Statics (CATS). In *2012 ASEE Annual Conference & Exposition Proceedings*. ASEE Conferences, San Antonio, Texas, 25.1457.1–25.1457.19. https://doi.org/10.18260/1-2--22214

[12] Travis Desell. 2013. Using Actors and the SALSA Programming Language to Introduce Concurrency in Computer Science II. In *2013 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum*. IEEE, 1257–1262. https://doi.org/10.1109/IPDPSW.2013.153

[13] Benedict du Boulay. 1986. Some Difficulties of Learning to Program Areas of Difficulty. *J. EDUCATIONAL COMPUTING RESEARCH* 2, 1 (1986), 57–73. https://doi.org/10.2190/3LFX-9RRF-67T8-UVK9

[14] Kathi Fisler, Shriram Krishnamurthi, and Preston Tunnell Wilson. 2017. Assessing and Teaching Scope, Mutation, and Aliasing in Upper-Level Undergraduates. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE '17)*. Association for Computing Machinery, New York, NY, USA, 213–218. https://doi.org/10.1145/3017680.3017777

[15] Eric Fouh, Ville Karavirta, Daniel A. Breakiron, Sally Hamouda, Simin Hall, Thomas L. Naps, and Clifford A. Shaffer. 2014. Design and architecture of an interactive eTextbook The OpenDSA system. *Science of Computer Programming* 88, 1 (2014), 22–40. https://doi.org/10.1016/j.scico.2013.11.040

[16] Daniela Giordano, Francesco Maiorana, Andrew Paul Csizmadia, Simon Marsden, Charles Riedesel, Shitanshu Mishra, and Lina Vinikienundefined. 2015. New Horizons in the Assessment of Computer Science at School and Beyond: Leveraging on the ViVA Platform. In *Proceedings of the 2015 ITiCSE on Working Group Reports (ITICSE-WGR '15)*. Association for Computing Machinery, New York, NY, USA, 117–147. https://doi.org/10.1145/2858796.2858801

[17] Ken Goldman, Paul Gross, Cinda Heeren, Geoffrey Herman, Lisa Kaczmarczyk, Michael C. Loui, and Craig Zilles. 2008. Identifying Important and Difficult

[4]https://groups.google.com/forum/#!forum/cs2-bdsi-concept-inventory

Concepts in Introductory Computing Courses Using a Delphi Process. *SIGCSE Bull.* 40, 1 (March 2008), 256–260. https://doi.org/10.1145/1352322.1352226

[18] Ken Goldman, Paul Gross, Cinda Heeren, Geoffrey L. Herman, Lisa Kaczmarczyk, Michael C. Loui, and Craig Zilles. 2010. Setting the Scope of Concept Inventories for Introductory Computing Subjects. *ACM Trans. Comput. Educ.* 10, 2, Article 5 (June 2010), 29 pages. https://doi.org/10.1145/1789934.1789935

[19] Philip J. Guo. 2013. Online Python Tutor: Embeddable Web-Based Program Visualization for Cs Education. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE '13)*. Association for Computing Machinery, New York, NY, USA, 579–584. https://doi.org/10.1145/2445196.2445368

[20] Mark Guzdial, Shriram Krishnamurthi, Juha Sorva, and Jan Vahrenhold. 2019. Notional Machines and Programming Language Semantics in Education (Dagstuhl Seminar 19281). *Dagstuhl Reports* 9, 7 (2019), 1–23. https://doi.org/10.4230/DagRep.9.7.1

[21] Sally Hamouda, Stephen H Edwards, Hicham G Elmongui, Jeremy V Ernst, and Clifford A Shaffer. 2017. A basic recursion concept inventory. *Computer Science Education* 27, 2 (2017), 121–148. https://doi.org/10.1080/08993408.2017.1414728

[22] Brian Harrington and Nick Cheng. 2018. Tracing vs. Writing Code: Beyond the Learning Hierarchy. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education (SIGCSE '18)*. Association for Computing Machinery, New York, NY, USA, 423–428. https://doi.org/10.1145/3159450.3159530

[23] Richard Herriot. 2019. What Kind of Research is Research Through Design?. In *IASDR 2019 Conference Proceedings*. International Association of Societies of Design Research, Manchester, 11. https://iasdr2019.org/uploads/files/Proceedings/op-f-1078-Her-R.pdf

[24] Matthew Hertz and Maria Jump. 2013. Trace-based Teaching in Early Programming Courses. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE '13)*. ACM, New York, NY, USA, 561–566. https://doi.org/10.1145/2445196.2445364

[25] Simon Holland, Robert Griffiths, and Mark Woodman. 1997. Avoiding Object Misconceptions. *SIGCSE Bull.* 29, 1 (March 1997), 131–134. https://doi.org/10.1145/268085.268132

[26] Cruz Izu, Carsten Schulte, Ashish Aggarwal, Quintin Cutts, Rodrigo Duran, Mirela Gutica, Birte Heinemann, Eileen Kraemer, Violetta Lonati, Claudio Mirolo, and Renske Weeda. 2019. Fostering Program Comprehension in Novice Programmers - Learning Activities and Learning Trajectories. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education (ITiCSE-WGR '19)*. Association for Computing Machinery, New York, NY, USA, 27–52. https://doi.org/10.1145/3344429.3372501

[27] Joint Task Force on Computing Curricula, Association for Computing Machinery (ACM) and IEEE Computer Society. 2013. *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science*. Association for Computing Machinery, New York, NY, USA.

[28] Natalie Jorion, Brian D. Gane, Katie James, Lianne Schroeder, Louis V. DiBello, and James W. Pellegrino. 2015. An Analytic Framework for Evaluating the Validity of Concept Inventory Claims. *Journal of Engineering Education* 104, 4 (oct 2015), 454–496. https://doi.org/10.1002/jee.20104

[29] Lisa C. Kaczmarczyk, Elizabeth R. Petrick, J. Philip East, and Geoffrey L. Herman. 2010. Identifying Student Misconceptions of Programming. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education (SIGCSE '10)*. Association for Computing Machinery, New York, NY, USA, 107–111. https://doi.org/10.1145/1734263.1734299

[30] Michael T. Kane. 2013. Validating the Interpretations and Uses of Test Scores. *Journal of Educational Measurement* 50, 1 (mar 2013), 1–73. https://doi.org/10.1111/jedm.12000

[31] Kuba Karpierz and Steven A. Wolfman. 2014. Misconceptions and Concept Inventory Questions for Binary Search Trees and Hash Tables. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education (SIGCSE '14)*. Association for Computing Machinery, New York, NY, USA, 109–114. https://doi.org/10.1145/2538862.2538902

[32] Cazembe Kennedy and Eileen T. Kraemer. 2018. What Are They Thinking?: Eliciting Student Reasoning About Troublesome Concepts in Introductory Computer Science. In *Proceedings of the 18th Koli Calling International Conference on Computing Education Research (Koli Calling '18)*. ACM, New York, NY, USA, Article 7, 10 pages. https://doi.org/10.1145/3279720.3279728

[33] Yifat Ben-David Kolikant. 2001. Gardeners and Cinema Tickets: High School Students' Preconceptions of Concurrency. *Computer Science Education* 11, 3 (2001), 221–245.

[34] Yifat Ben-David Kolikant. 2004. Learning concurrency: evolution of students' understanding of synchronization. *International Journal of Human-Computer Studies* 60, 2 (2004), 243–268. https://doi.org/10.1016/j.ijhcs.2003.10.005

[35] Yifat Ben-David Kolikant. 2005. Students' Alternative Standards for Correctness. In *Proceedings of the First International Workshop on Computing Education Research (ICER '05)*. ACM, New York, NY, USA, 37–43. https://doi.org/10.1145/1089786.1089790

[36] Ari Korhonen. 2010. Applications of Visual Algorithm Simulation. In *Handbook of Research on Discrete Event Simulation Environments: Technologies and Applications*, Evon M. O. Abu-Taieh and Asim A. El-Sheikh (Eds.). IGI Global, Hershey, PA, USA, 234–251. https://doi.org/10.4018/978-1-60566-774-4

[37] Sophia Krause-Levy, Sander Valstar, Leo Porter, and William G. Griswold. 2020. Exploring the Link Between Prerequisites and Performance in Advanced Data Structures. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education (SIGCSE '20)*. Association for Computing Machinery, New York, NY, USA, 386–392. https://doi.org/10.1145/3328778.3366867

[38] Einari Kurvinen, Niko Hellgren, Erkki Kaila, Mikko-Jussi Laakso, and Tapio Salakoski. 2016. Programming Misconceptions in an Introductory Level Programming Course Exam. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '16)*. Association for Computing Machinery, New York, NY, USA, 308–313. https://doi.org/10.1145/2899415.2899447

[39] Thomas Lancaster, Anthony Robins, and Sally A. Fincher. 2019. Assessment and plagiarism. In *The Cambridge handbook of computing education research*, Sally A Fincher and Anthony V Robins (Eds.). Cambridge University Press, Cambridge, UK, 414–444.

[40] Catherine Legg and Christopher Hookway. 2020. Pragmatism. In *The Stanford Encyclopedia of Philosophy* (fall 2020 ed.), Edward N. Zalta (Ed.). Metaphysics Research Lab, Stanford University, Standford, CA, USA.

[41] Jacqueline P. Leighton and Mark J. Gierl. 2007. Verbal reports as data for cognitive diagnostic assessment. In *Cognitive Diagnostic Assessment for Education: Theory and Applications*, Jacqueline Leighton and Mark Gierl (Eds.). Cambridge University Press, Cambridge, 146–172. https://doi.org/10.1017/CBO9780511611186.006

[42] Gary Lewandowski, Dennis J. Bouvier, Robert McCartney, Kate Sanders, and Beth Simon. 2007. Commonsense Computing (Episode 3): Concurrency and Concert Tickets. In *Proceedings of the Third International Workshop on Computing Education Research (ICER '07)*. Association for Computing Machinery, New York, NY, USA, 133–144. https://doi.org/10.1145/1288580.1288598

[43] Barbara Liskov and John Guttag. 2000. *Program development in JAVA: abstraction, specification, and object-oriented design*. Pearson Education.

[44] Raymond Lister, Elizabeth S. Adams, Sue Fitzgerald, William Fone, John Hamer, Morten Lindholm, Robert McCartney, Jan Erik Moström, Kate Sanders, Otto Seppälä, Beth Simon, and Lynda Thomas. 2004. A Multi-National Study of Reading and Tracing Skills in Novice Programmers. In *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education (ITiCSE-WGR '04)*. Association for Computing Machinery, New York, NY, USA, 119–150. https://doi.org/10.1145/1044550.1041673

[45] Jan Lönnberg, Anders Berglund, and Lauri Malmi. 2009. How Students Develop Concurrent Programs. In *Proceedings of the Eleventh Australasian Conference on Computing Education - Volume 95 (ACE '09)*. Australian Computer Society, Inc., Darlinghurst, Australia, 129–138. http://dl.acm.org/citation.cfm?id=1862712.1862732

[46] Mike Lopez, Jacqueline Whalley, Phil Robbins, and Raymond Lister. 2008. Relationships between Reading, Tracing and Writing Skills in Introductory Programming. In *Proceedings of the Fourth International Workshop on Computing Education Research (ICER '08)*. Association for Computing Machinery, New York, NY, USA, 101–112. https://doi.org/10.1145/1404520.1404531

[47] Andrew Luxton-Reilly, Brett A. Becker, Yingjun Cao, Roger McDermott, Claudio Mirolo, Andreas Mühling, Andrew Petersen, Kate Sanders, Simon, and Jacqueline Whalley. 2018. Developing Assessments to Determine Mastery of Programming Fundamentals. In *Proceedings of the 2017 ITiCSE Conference on Working Group Reports (ITiCSE-WGR '17)*. Association for Computing Machinery, New York, NY, USA, 47–69. https://doi.org/10.1145/3174781.3174784

[48] Andrew Luxton-Reilly, Simon, Ibrahim Albluwi, Brett A. Becker, Michail Giannakos, Amruth N. Kumar, Linda Ott, James Paterson, Michael James Scott, Judy Sheard, and Claudia Szabo. 2018. Introductory Programming: A Systematic Literature Review. In *Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE 2018 Companion)*. Association for Computing Machinery, New York, NY, USA, 55–106. https://doi.org/10.1145/3293881.3295779

[49] Jan Lönnberg and Anders Berglund. 2007. Students' Understandings of Concurrent Programming. In *Proceedings of the Seventh Baltic Sea Conference on Computing Education Research - Volume 88*. Australian Computer Society, Inc., Darlinghurst, Australia, 77–86. http://dl.acm.org/citation.cfm?id=2449323.2449332

[50] Susan McKenney and Thomas C. Reeves. 2014. Design and development research. In *Handbook of Research on Educational Communications and Technology: Fourth Edition*. Springer New York, New York, NY, 131–140. https://doi.org/10.1007/978-1-4614-3185-5_11

[51] Susan E. McKenney and Thomas C. Reeves. 2012. *Conducting Educational Design Research*. Routledge, London.

[52] Bertrand Meyer. 1995. Static Typing. *SIGPLAN OOPS Mess.* 6, 4 (Oct. 1995), 20–29. https://doi.org/10.1145/260111.260214

[53] Bertrand Meyer. 1997. *Object-oriented software construction*. Vol. 2. Prentice hall Englewood Cliffs, USA.

[54] Thomas L. Naps, Guido Rößling, Vicki Almstrum, Wanda Dann, Rudolf Fleischer, Chris Hundhausen, Ari Korhonen, Lauri Malmi, Myles McNally, Susan Rodger, and J. Ángel Velázquez-Iturbide. 2002. Exploring the Role of Visualization and Engagement in Computer Science Education. In *Working Group Reports from*

ITiCSE on Innovation and Technology in Computer Science Education (ITiCSE-WGR '02). Association for Computing Machinery, New York, NY, USA, 131–152. https://doi.org/10.1145/960568.782998

[55] Greg L. Nelson, Andrew Hu, Benjamin Xie, and Amy J. Ko. 2019. Towards Validity for a Formative Assessment for Language-specific Program Tracing Skills. In *Proceedings of the 19th Koli Calling International Conference on Computing Education Research (Koli Calling '19)*. ACM, New York, NY, USA, Article 20, 10 pages. https://doi.org/10.1145/3364510.3364525

[56] Greg L. Nelson and Andrew J. Ko. 2018. On Use of Theory in Computing Education Research. In *Proceedings of the 2018 ACM Conference on International Computing Education Research (ICER '18)*. Association for Computing Machinery, New York, NY, USA, 31–39. https://doi.org/10.1145/3230977.3230992

[57] Greg L. Nelson, Benjamin Xie, and Andrew J. Ko. 2017. Comprehension First: Evaluating a Novel Pedagogy and Tutoring System for Program Tracing in CS1. In *Proceedings of the 2017 ACM Conference on International Computing Education Research (ICER '17)*. Association for Computing Machinery, New York, NY, USA, 2–11. https://doi.org/10.1145/3105726.3106178

[58] John Ousterhout. 2018. *A Philosophy of Software Design*. Yaknyam Press.

[59] Miranda C. Parker, Mark Guzdial, and Shelly Engleman. 2016. Replication, Validation, and Use of a Language Independent CS1 Knowledge Assessment. In *Proceedings of the 2016 ACM Conference on International Computing Education Research (ICER '16)*. Association for Computing Machinery, New York, NY, USA, 93–101. https://doi.org/10.1145/2960310.2960316

[60] Wolfgang Paul and Jan Vahrenhold. 2013. Hunting High and Low: Instruments to Detect Misconceptions Related to Algorithms and Data Structures. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE '13)*. Association for Computing Machinery, New York, NY, USA, 29–34. https://doi.org/10.1145/2445196.2445212

[61] Roy D Pea. 1986. Language-Independent Conceptual "Bugs" in Novice Programming. *Journal of Educational Computing Research* 2, 1 (Feb. 1986), 25–36.

[62] Thomas Pelchen, Luke Mathieson, and Raymond Lister. 2020. On the Evidence for a Learning Hierarchy in Data Structures Exams. In *Proceedings of the Twenty-Second Australasian Computing Education Conference (ACE'20)*. Association for Computing Machinery, New York, NY, USA, 122–131. https://doi.org/10.1145/3373165.3373179

[63] D. N. Perkins and Fay Martin. 1986. Fragile Knowledge and Neglected Strategies in Novice Programmers. In *Papers Presented at the First Workshop on Empirical Studies of Programmers on Empirical Studies of Programmers*. Ablex Publishing Corp., Norwood, NJ, USA, 213–229. http://dl.acm.org/citation.cfm?id=21842.28896

[64] Yakov Persky and Mordechai Ben-Ari. 1998. Re-engineering a Concurrency Simulator. *SIGCSE Bull.* 30, 3 (Aug. 1998), 185–188. https://doi.org/10.1145/290320.283117

[65] Leo Porter, Daniel Zingaro, Cynthia Lee, Cynthia Taylor, Kevin C. Webb, and Michael Clancy. 2018. Developing Course-Level Learning Goals for Basic Data Structures in CS2. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education (SIGCSE '18)*. Association for Computing Machinery, New York, NY, USA, 858–863. https://doi.org/10.1145/3159450.3159457

[66] Leo Porter, Daniel Zingaro, Soohyun Nam Liao, Cynthia Taylor, Kevin C. Webb, Cynthia Lee, and Michael Clancy. 2019. BDSI: A Validated Concept Inventory for Basic Data Structures. In *Proceedings of the 2019 ACM Conference on International Computing Education Research (ICER '19)*. Association for Computing Machinery, New York, NY, USA, 111–119. https://doi.org/10.1145/3291279.3339404

[67] Yizhou Qian and James Lehman. 2017. Students' Misconceptions and Other Difficulties in Introductory Programming: A Literature Review. *ACM Trans. Comput. Educ.* 18, 1, Article 1 (Oct. 2017), 24 pages. https://doi.org/10.1145/3077618

[68] Noa Ragonis and Mordechai Ben-Ari. 2005. A long-term investigation of the comprehension of OOP concepts by novices. *Computer Science Education* 15, 3 (2005), 203–221. https://doi.org/10.1080/08993400500224310

[69] Thomas C. Reeves. 2006. Design research from the technology perspective. In *Educational design research*. Routledge, 86–109.

[70] Gerard Rowe and Chris Smaill. 2007. Development of an electromagnetic course—concept inventory—a work in progress. In *Proceedings of the eighteenth Conference of Australian Association for Engineering*. Department of Computer Science and Software Engineering, The University of Melbourne, Melbourne, Australia, 7.

[71] Jorma Sajaniemi, Marja Kuittinen, and Taina Tikansalo. 2008. A Study of the Development of Students' Visualizations of Program State during an Elementary Object-Oriented Programming Course. *J. Educ. Resour. Comput.* 7, 4, Article 3 (Jan. 2008), 31 pages. https://doi.org/10.1145/1316450.1316453

[72] Kate Sanders, Marzieh Ahmadzadeh, Tony Clear, Stephen H. Edwards, Mikey Goldweber, Chris Johnson, Raymond Lister, Robert McCartney, Elizabeth Patitsas, and Jaime Spacco. 2013. The Canterbury QuestionBank: Building a Repository of Multiple-Choice CS1 and CS2 Questions. In *Proceedings of the ITiCSE Working Group Reports Conference on Innovation and Technology in Computer Science Education-Working Group Reports (ITiCSE -WGR '13)*. Association for Computing Machinery, New York, NY, USA, 33–52. https://doi.org/10.1145/2543882.2543885

[73] David Sands, Mark Parker, Holly Hedgeland, Sally Jordan, and Ross Galloway. 2018. Using concept inventories to measure understanding. *Higher Education Pedagogies* 3, 1 (jan 2018), 173–182. https://doi.org/10.1080/23752696.2018.1433546

[74] Aidsa I. Santiago Roman. 2009. *Fitting cognitive diagnostic assessment to the Concept Assessment Tool for Statics (CATS)*. Ph.D. Dissertation. University of Washington. https://www.proquest.com/docview/304991523?accountid=14784 Copyright - Database copyright ProQuest LLC; ProQuest does not claim copyright in the individual underlying works; Last updated - 2020-10-07.

[75] Clifford A. Shaffer, Ville Karavirta, Ari Korhonen, and Thomas L. Naps. 2011. OpenDSA: Beginning a Community Active-eBook Project. In *Proceedings of the 11th Koli Calling International Conference on Computing Education Research, Joensuu, Finland, November 17-20, 2011*. University of Eastern Finland, Joensuu, Finland, 112–117.

[76] Simon, Judy Sheard, Daryl D'Souza, Peter Klemperer, Leo Porter, Juha Sorva, Martijn Stegeman, and Daniel Zingaro. 2016. Benchmarking Introductory Programming Exams: How and Why. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '16)*. Association for Computing Machinery, New York, NY, USA, 154–159. https://doi.org/10.1145/2899415.2899473

[77] Simon, Judy Sheard, Daryl D'Souza, Peter Klemperer, Leo Porter, Juha Sorva, Martijn Stegeman, and Daniel Zingaro. 2016. Benchmarking Introductory Programming Exams: Some Preliminary Results. In *Proceedings of the 2016 ACM Conference on International Computing Education Research (ICER '16)*. Association for Computing Machinery, New York, NY, USA, 103–111. https://doi.org/10.1145/2960310.2960337

[78] Beth Simon, Mike Clancy, Robert McCartney, Briana Morrison, Brad Richards, and Kate Sanders. 2010. Making Sense of Data Structures Exams. In *Proceedings of the Sixth International Workshop on Computing Education Research (ICER '10)*. Association for Computing Machinery, New York, NY, USA, 97–106. https://doi.org/10.1145/1839594.1839612

[79] Eric Snow, Daisy Rutstein, Marie Bienkowski, and Yuning Xu. 2017. Principled Assessment of Student Learning in High School Computer Science. In *Proceedings of the 2017 ACM Conference on International Computing Education Research (ICER '17)*. Association for Computing Machinery, New York, NY, USA, 209–216. https://doi.org/10.1145/3105726.3106186

[80] Juha Sorva. 2007. Students' Understandings of Storing Objects. In *Proceedings of the Seventh Baltic Sea Conference on Computing Education Research - Volume 88 (Koli Calling '07)*. Australian Computer Society, Inc., AUS, 127–135.

[81] Juha Sorva. 2012. *Visual program simulation in introductory programming education; Visuaalinen ohjelmasimulaatio ohjelmoinnin alkeisopetuksessa*. G4 Monografiaväitöskirja. Aalto-yliopisto. http://urn.fi/URN:ISBN:978-952-60-4626-6

[82] Juha Sorva. 2013. Notional Machines and Introductory Programming Education. *Trans. Comput. Educ.* 13, 2 (Jul 2013), 8:1–8:31. https://doi.org/10.1145/2483710.2483713

[83] Juha Sorva, Ville Karavirta, and Lauri Malmi. 2013. A Review of Generic Program Visualization Systems for Introductory Programming Education. *Trans. Comput. Educ.* 13, 4 (Nov 2013), 15:1–15:64. https://doi.org/10.1145/2490822

[84] Juha Sorva, Jan Lönnberg, and Lauri Malmi. 2013. Students' ways of experiencing visual program simulation. *Computer Science Education* 23, 3 (Sept. 2013), 207–238. https://doi.org/10.1080/08993408.2013.807962

[85] Milton E. Strauss and Gregory T. Smith. 2009. Construct validity: Advances in theory and methodology. *Annual Review of Clinical Psychology* 5 (4 2009), 1–25. https://doi.org/10.1146/annurev.clinpsy.032408.153639

[86] Filip Strömbäck, Linda Mannila, Mikael Asplund, and Mariam Kamkar. 2019. A Student's View of Concurrency - A Study of Common Mistakes in Introductory Courses on Concurrency. In *Proceedings of the 2019 ACM Conference on International Computing Education Research (ICER '19)*. Association for Computing Machinery, New York, NY, USA, 229–237. https://doi.org/10.1145/3291279.3339415

[87] C. Taylor, D. Zingaro, L. Porter, K.C. Webb, C.B. Lee, and M. Clancy. 2014. Computer science concept inventories: past and future. *Computer Science Education* 24, 4 (2014), 253–276. https://doi.org/10.1080/08993408.2014.970779

[88] Allison Elliott Tew and Brian Dorn. 2013. The Case for Validated Tools in Computer Science Education Research. *Computer* 46, 9 (sep 2013), 60–66. https://doi.org/10.1109/MC.2013.259

[89] Sander Valstar, William G. Griswold, and Leo Porter. 2019. The Relationship between Prerequisite Proficiency and Student Performance in an Upper-Division Computing Course. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE '19)*. Association for Computing Machinery, New York, NY, USA, 794–800. https://doi.org/10.1145/3287324.3287419

[90] Peter Wegner. 1990. Concepts and Paradigms of Object-Oriented Programming. *SIGPLAN OOPS Mess.* 1, 1 (Aug. 1990), 7–87. https://doi.org/10.1145/382192.383004

[91] Jacqueline L. Whalley, Raymond Lister, Errol Thompson, Tony Clear, Phil Robbins, P. K. Ajith Kumar, and Christine Prasad. 2006. An Australasian Study of Reading and Comprehension Skills in Novice Programmers, Using the Bloom and SOLO Taxonomies. In *Proceedings of the 8th Australasian Conference on Computing Education - Volume 52 (ACE '06)*. Australian Computer Society, Inc., AUS, 243–252.

[92] Benjamin Xie, Dastyni Loksa, Greg L. Nelson, Matthew J. Davidson, Dongsheng Dong, Harrison Kwik, Alex Hui Tan, Leanne Hwa, Min Li, and Andrew J. Ko. 2019. A theory of instruction for introductory programming skills. *Computer Science Education* 29, 2-3 (2019), 205–253. https://doi.org/10.1080/08993408.2019.1565235

[93] Benjamin Xie, Greg L. Nelson, and Andrew J. Ko. 2018. An Explicit Strategy to Scaffold Novice Program Tracing. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education (SIGCSE '18)*. Association for Computing Machinery, New York, NY, USA, 344–349. https://doi.org/10.1145/3159450.3159527

CONTENTS

Note that the numbering here reflects the numbering of the questions. M is for Modified questions and O is for Other questions (that were not modified).

A STUDY OF PREREQUISITE SKILLS

In order to better connect our qualitative coding categories to existing knowledge, and as a way of validating our categories, we associated each of our codes (Section 4.1, Tables 1 to 6) with topics in the ACM Computing Curriculum Guidelines [2], prior work by Goldman et al. [18] to identify prerequisite topics, and Misconception Catalogue compiled by Sorva [81].

A.1 ACM CC2013

The ACM 2013 Curriculum Guide [27] consists of 18 Knowledge Areas in computing.

- Algorithms and Complexity (AL)
- Architecture and Organization (AR)
- Computational Science (CN)
- Discrete Structures (DS)
- Graphics and Visualization (GV)
- Human-Computer Interaction (HCI)
- Information Assurance and Security (IAS)
- Information Management (IM)
- Intelligent Systems (IS)
- Networking and Communications (NC)
- Operating Systems (OS)
- Platform-based Development (PBD)
- Parallel and Distributed Computing (PD)
- Programming Languages (PL)
- Software Development Fundamentals (SDF)
- Software Engineering (SE)
- Systems Fundamentals (SF)
- Social Issues and Professional Practice (SP)

These Knowledge Areas correspond to particular courses or course sequences in many programs. Therefore, the Curriculum Guide can be seen as an enumeration of topics typically taught in various CS courses. Given the fact that many computer science programs as a part of accreditation align their courses whenever possible to the 2013 Curriculum Guide, the Knowledge Areas can be seen as reflecting skills commonly taught in CS1 courses.

This raises questions such as how these KAs are related to each other. We need to realize that the KAs above are interconnected. Concepts in one KA may build upon another KA. The reader is encouraged to read the CC2013 and the Body of Knowledge as a whole rather than focusing on any given Knowledge Area in isolation.

The Body of Knowledge is a specification of the content to be covered and a curriculum is an implementation for it. However, Computer Science, unlike many technical disciplines, does not have a well-described list of topics that appear in virtually all introductory courses [2]. Many of them focus on topics such as Software Development Fundamentals, Programming Languages, and Software Engineering. Some courses start with object-oriented programming, while others use functional programming. In addition, it is not said that all Software Development Fundamentals should be covered in a first course. In practice, however, most fundamental topics are typically covered in CS1.

Not all technically relevant concepts to a computer scientist (programming, software processes, algorithms, data structure, abstraction, performance, security, concurrency, etc.), even their early introduction, can come in the first course. Many topics will appear only in advanced courses. Institutions make their own decisions on which topics to select for advanced courses, and which are considered prerequisite skills taught in introductory courses. This also includes software design and development best practices, such as unit testing and programming patterns, as well as tools used in teaching such as version control systems, and industrial integrated development environments (IDEs). Thus, in this report, prerequisites are skills that are relative to the choices made for the specification of the content to be covered and the curriculum that is an implementation for it. In addition, not all skills and knowledge covered in an introductory course will be prerequisites for all advanced courses. If there is no demand for object-oriented programming in an advanced course, the OO concepts are not prerequisite knowledge even though the introductory course was designed in an objects-first approach.

As said earlier, the Body of Knowledge can be interpreted as a specification of the content to be covered and a curriculum is an implementation of it. Many curricula meet the specification. However, the above knowledge areas are not intended to be in one-to-one correspondence with particular courses in a curriculum. In addition, a curriculum should have courses that incorporate topics from multiple Knowledge Areas. CC2013 identifies topics as “Core” or “Elective”, with the core further subdivided into “Tier-1” and “Tier-2”. A curriculum should include all topics in the Tier-1 core and ensure that all students cover this material. However, the reader must note that even most of the topics within a Tier-1 are such that they are taught in advanced courses. For example, AL/Basic Analysis has the following Core-Tier-1:

- Differences among best, expected, and worst case behaviors of an algorithm
- Asymptotic analysis of upper and expected complexity bounds
- Big O notation: formal definition

- Complexity classes, such as constant, logarithmic, linear, quadratic, and exponential
- Empirical measurements of performance
- Time and space trade-offs in algorithms

As we can see, many of these topics are taught in a course called Data Structures and Algorithms, or similar, which also has a prerequisite course (e.g., CS1 or CS2). Thus, we are not going to list all CC2013 Knowledge Areas in our paper, but review the areas of CC2013 closest to the tracing-related prerequisite skills: SDF/Fundamental Programming Concepts, SDF/Fundamental Data Structures, and SDF/Development Methods as well as PL/Object-Oriented Programming, and PL/Basic Type Systems. For brevity, here we only give an examples of SDF/Development Methods. It is expected that every curriculum should invest 10 core Tier-1 hours for this. An "hour" corresponds to the time required to present the material in a traditional lecture-oriented format. However, the hour count does not include any additional work that is associated with a lecture (e.g., in self-study, laboratory sessions, and assessments). According to CC2013, the SDF/Development Methods should include,

- Program comprehension,
- Program correctness,
  - Types of errors (syntax, logic, run-time)
  - The concept of a specification,
  - Defensive programming (e.g., secure coding, exception handling),
  - Code reviews,
  - Testing fundamentals and test-case generation,
  - The role and the use of contracts, including pre- and post-conditions, and
  - Unit testing.
- Simple refactoring,
- Modern programming environments,
  - Code search,
  - Programming using library components and their APIs.
- Debugging strategies, and
- Documentation and program style.

As we can see, even these topics are described at such a high level of abstraction that there must exist many underlying lower level concepts (operators, variables, assignments, loop constructs, conditional branching, subroutines, etc.) required to master the whole knowledge area.

CC2013 also has examples of Learning Outcomes (LO) related to each Knowledge Area. The following are good examples of LOs for SDF/Development Methods that we are investigating in this report:

- Trace the execution of a variety of code segments and write summaries of their computations.
- Construct, execute and debug programs using a modern IDE and associated tools such as unit testing tools and visual debuggers.
- Construct and debug programs using the standard libraries available with a chosen programming language.

A.2 Core Concepts Identified by Experts

Goldman et al. [17, 18] set out to create a concept inventory for introductory computing subjects. An important part of this process is to investigate which core concepts are typically covered in introductory courses, and which of those are perceived to be important and difficult. This allows the final concept inventory to focus on the most important concepts that students are most likely to find difficult, thus keeping the size of the concept inventory down.

We present a summary of the final concepts for programming fundamentals below as that is the subject most relevant to this report. The remaining concepts can be found in the original paper [18]. To find the most difficult and important concepts, the authors represented each concept as a point on a 2D plane, the x coordinate being the mean importance, and the y coordinate being the mean difficulty. The concepts closest to the maximum point $(10, 10)$ were then deemed to be the most important and difficult topics. The top 11 such topics are marked with an asterisk below.

PA1 Parameters/Arguments I: Understanding the difference between "Call by Reference" and "Call by Value".

PA2 Parameters/Arguments II: Understanding the difference between "Formal Parameters" and "Actual Parameters".

PA3* Parameters/Arguments III: Understanding the scope of parameters, correctly using parameters in procedure design.

PROC* Procedures/Functions/Methods: (e.g., designing and declaring procedures, choosing parameters and return values, properly invoking procedures)

CF Control Flow: Correctly tracing code through a given model of execution.

TYP Types: (e.g., choosing appropriate types for data, reasoning about primitive and object types, understanding type implications in expressions (e.g., integer division rather than floating point))

BOOL Boolean Logic: (e.g., constructing and evaluating boolean expressions, using them appropriately in the design of conditionals and return expressions)

COND Conditionals: (e.g., writing correct expressions for conditions, tracing execution through nested conditional structures correctly)

SVS Syntax vs. Semantics: Understanding the difference between a textual code segment and its overarching purpose and operation.

OP Operator Precedence: (e.g., writing and evaluating expressions using multiple operators)

AS Assignment Statements: (e.g., interpreting the assignment operator not as the comparison operator, assigning values from the right hand side of the operator to the left hand side of the operator, understanding the difference between assignment and a mathematical statement of equality)

SCO* Scope: (e.g., understanding the difference between local and global variables and knowing when to choose which type, knowing declaration must occur before usage, masking, implicit targets (e.g., this operator in Java))

CO Classes and Objects: Understanding the separation between definition and instantiation.

SCDE Scope Design: (e.g., understanding difference in scope between fields and local variables, appropriately using visibility properties of fields and methods, encapsulation)

INH* Inheritance: (e.g., understanding the purpose of extensible design and can use it)

POLY Polymorphism: (e.g., understanding and using method dispatch capabilities, knowing how to use general types for extensibility)

STAM Static Variables and Methods: (e.g., understanding and using methods and variables of a class which are not invoked on or accessed by an instance of the class)

PVR Primitive and Reference Type Variables: Understanding the difference between variables which hold data and variables which hold memory references/pointers.

APR* Abstractions/Pattern Recognition and Use: (e.g., translating the structure of a solution to the solution of another similar problem)

IT1 Iteration/Loops I: Tracing the execution of nested loops correctly.

IT2 Iteration/Loops II: Understanding that loop variables can be used in expressions that occur in the body of a loop.

REC* Recursion: (e.g., tracing execution of recursive procedures, can identofy recursive patterns and translate into recursive structures)

AR1 Arrays I: Identifying and handling off-by-one errors when using in loop structures.

AR2 Arrays II: Understanding the difference between a reference to an array and an element of an array.

AR3 Arrays III: Understanding the declaration of an array and correctly manipulating arrays.

MMR* Memory Model/Reference/Pointers: (e.g., understanding the connection between high-level language concepts and the underlying memory model, visualizing memory references, correct use of reference parameters, indirection, and manipulation of pointer-based data structures)

DPS1* Design and Problem Solving I: Understands and uses functional decomposition and modularization: solutions are not one long procedure.

DPS2* Design and Problem Solving II: Ability to identify characteristics of a problem and formulate a solution design.

DEH* Debugging/Exception Handling: (e.g., deceloping and using practices for finding code errors)

IVI Interface fs. Implementation: (e.g., understanding the difference between the design of a type and the design of its implementation)

IAC Interfaces and Abstract Classes: (e.g., understanding general types in design, designing extensible systems, ability to design around such abstract types)

DT* Designing Tests: (e.g., ability to design tests that effectively cover a specification)

The authors characterized concepts with a high standard deviation of rankings into two types: outlier and controversial. Outlier concepts (PA1, IT2, TYP, PVR, REC) had a strong consensus but one or two outliers. Controversial concepts (INH, MMR), on the other hand, had clustering around two ratings. The authors theorize that this might partially be due to different experts teaching different programming languages in CS1.

A.3 Misconception Catalogue

Misconceptions can be caused by a lack of knowledge of the syntax, not knowing how a particular syntactical construct behaves (i.e., due to an incorrect or incomplete notional machine [13, 82]), or not knowing how to use a particular construct in order to solve a programming problem (i.e. lacking strategic knowledge). Several misconceptions in introductory programming have been identified and addressed in the literature [38, 61, 67, 82]. For example, a classic syntactical misconception is the use of an assignment operator (=) instead of the comparison operator (==). At a conceptual level (notional machine), an example of a misconception is that a variable can hold more than one value; this is manifested in the task of swapping two variables. With respect to the strategic level, novices find modularization and decomposition, general abstraction, testing, and debugging very difficult [17].

In this report we do not contribute new misconceptions but instead use existing knowledge about misconceptions to check that our new assessment questions might feasibly catch frequent misconceptions. We wanted to check how misconceptions from the research literature aligned with our prerequisite skills coding. However, as the number of studies related to misconceptions is too large to be cited her, we chose to use the Misconception Catalogue collected by Sorva. It represents a review of literature from the past forty years [81]. Although, it gives examples of novice programmers' misconceptions about the content of introductory programming courses in general, it is somewhat leaning towards misconceptions found in courses taking Object-Oriented approach.

In the Misconceptions Catalogue, the topics of misconceptions are grouped into the following structure:

(1) General (the overall nature of programs and program execution),
(2) VarAssign (variables, assignment and expression evaluation),
(3) Control (flow of control, selection and interation),
(4) Calls (subprogram invocations and parameter passing),
(5) Rec (recursion),
(6) Refs (references and pointers, reference assignment and object identity),
(7) ObjClass (the object–class relationship and instantiation),
(8) ObjState (object state and attributes),
(9) Methods (issues specific to methods and methods calls),
(10) OtherOOP (other topics specific to object-oriented programming), and
(11) Misc (none of the above).

For examples of particular misconceptions mapped to our codebook, see Tables 1 to 6.

M MODIFIED QUESTIONS

M.1 Data Structure Question (Queue)

M.1.1 Original question. Consider the following data structure:

```java
public class Y<Key extends Comparable<Key>>
{
    private Key[] A = (Key[]) new Comparable[1];
    private int lo, hi, N;
    public void insert(Key in)
    {
        A[hi] = in;
        hi = hi + 1;
        if (hi == A.length) hi = 0;
        N = N + 1;
        if (N == A.length) rebuild();
    }
    public Key remove() // assumes Y is not empty
    {
        Key out = A[lo];
        A[lo] = null;
        lo = lo + 1;
        if (lo == A.length) lo = 0;
        N = N - 1;
        return out;
    }
    private void rebuild()
    {
        Key[] tmp =
            (Key[]) new Comparable[2*A.length];
        for (int i = 0; i < N; i++ )
            tmp[i] = A[(i + lo) % A.length];
        A = tmp;
        lo = 0;
        hi = N;
    }
}
```

(a) Class Y behaves like which well-known data structure?
A Stack
B Queue
C Priority queue
D Union-find

(b) Write the body of a method `int size()` that returns the number of elements in the data structure.
A `return N;`
B `return A.length;`
C `return A[N];`
D `return hi - lo;`

(c) Which invariant does the data structure maintain after every public operation?
A `N < A.length`
B `lo < hi`
C `hi < N`
D `hi == N`

(d) Draw the data structure (including the contents of `A` and the values of `hi`, `lo`, and `N`) after the following operations:

```java
Y y = new Y();
y.insert(1);
y.remove();
y.insert(2);
y.remove();
y.insert(3);
```

(e) How many array accesses does a single call to `Y.remove` take in the worst case? (To make this well-defined, we assume that the compiler performs no clever optimisations. That is, every array access we've written in the code will actually be performed.)
A $\sim 4N$
B 2
C $\sim 2N$
D 7

(f) How many array accesses does a single call to the most expensive public method of `Y` take in the worst case?
A linear in k.
B constant.
C linearithmic in k.
D quadratic in k.

(g) What is the number of array accesses per operation in the following sequence of $2k$ operations, starting from an empty data structure: `y.insert(1); y.remove(); y.insert(2); y.remove(); y.insert(3); y.remove(); ... y.insert(k); y.remove();`
A linear in k in the worst case and in the amortized case.
B constant in the worst case.
C constant in the amortized case, but linear in k in the worst case.
D quadratic in k in the worst case.

(h) True or false: The data structure `Y` uses space linear in `N`. Explain you answer on a separate piece of paper. (Be as formal *and short* as you can, but not shorter. If you use more than half a page of text you're on the wrong level of abstraction.)

M.1.2 Analysis of the original question. This question assesses the following skills categorized in this paper:

- Simple statements
- Operators
- Assignments
- Tracing
- Conditionals
- Loop constructs
- Array iteration
- Types
- Values and references
- Arrays
- Parameters
- Return values
- Function scoping and data flow
- Classes/records/ADT
- Reasoning about constraints
- Meta-tracing knowledge

The code implements a queue with a *circular* array and two integer: `lo` is the index of the last element in the queue and `hi` is the index just after the first element in the queue. Variable `N` represents the number of element in the queue. The array is rebuilt with doubled size whenever the insertion of an element exhausts the capacity of the array.

The correct answers for the first items are: B, A, A

More in-depth analysis on the original exercise:

- The answer to the first answer could be wrong for very different reasons:
  - you do not know these data structures, you are not able to differentiate them (advanced concept)
  - you do not understand the code
- If you know the difference between the mentioned data structures, a very general understanding of the code would be enough to answer item (a): you can exclude union-find because is a totally different setting; you can exclude priority queues because there are no comparisons; lines 8 and 17 should be enough to understand that insertion and removal occur in different places so the stack can also be excluded.
- If one does not understand the circular nature of this queue implementation, they might wrongly select option D for item (b) and option B for item (c). The relevant lines of code here are 9, 18, 27. However, the circularity is not explicitly assessed, since one could correctly answer to items (b) and (c) by considering only lines 10 and 19, without understanding the circularity.
- Item (c) addresses the possible confusion between the length of the array and the number of elements currently in the queue.
- To answer item (d) correctly you need to understand both the circularity and the rebuilding policy.
- To understand the rebuilding policy you need conditionals and array knowledge (line 11).
- Neither the circularity nature of the queue nor implementation and the rebuilding policy are assessed explicitly (separately).
- In item (d), extreme situations may occur: on the one hand, one can answer correctly by tracing the code line by line, without understanding what is going on on an abstract level, on the other hand, if one has already understood how the queue is implemented and they are able to reason about it at a high level, they could answer item (b) without considering the code at all.
- Items from (e) openly address advanced concepts (related to complexity) that still require referring to the code and considering its execution.

M.1.3 Summary of assessed skills. This exercise assesses the following advanced learning outcomes:

- knowledge: difference among different data structures; notion of worst case complexity and amortized complexity
- skills: understand a piece of code that implements a queue; analyse the complexity of an algorithm expressed in a piece of code

The prerequisite skills (from our code-book) required to solve this exercise are:

- Simple Statements
- Operators
- Assignments
- Tracing
- Conditionals
- Loop constructs
- Array iteration
- Arrays
- Type
- Return values
- Function scoping and data flow
- Classes/records/ADT
- Reasoning about constraints
- Meta-tracing knowledge

M.1.4 New version. (Changes are highlighted in bold)

(a) Class Y behaves like which well-known data structure?
A Stack
B Queue
C Priority queue
D Union find

(b) Write the body of a method `int size()` that returns the number of elements in the data structure.
A `return N;`
B `return A.length;`
C `return A[N];`
D `return hi - lo;`

(c) Which invariant does the data structure maintain after every public operation?
A `N < A.length`
B `lo < hi`
C `hi < N`
D `hi == N`

(c-1) Assume that:
`A` holds $\{3, 8, 4, 1\}$,
`lo` holds 3,
`hi` holds 2 and
`N` holds 2.
A Is the above situation something that can occur by calling a sequence of `insert` and `remove`? If yes, give such a sequence, otherwise explain why not.
B What are the contents of `A`, `lo` and `hi` after executing `rebuild`?

(d) Draw the data structure (including the contents of `A` and the values of `hi`, `lo`, and `N`) after the following operations, **and indicate how many times `rebuild` were called**:

```
Y y = new Y();
y.insert(1);
y.remove();
y.insert(2);
y.remove();
y.insert(3);
```

(d-1) What are the values of `a` and `b` after executing the following piece of code?

```
Y y = new Y();
Y z = new Z();
Y w = z;
w.insert(3);
z.insert(1);
y.insert(2);
```

```
int a = z.remove();
int b = y.remove();
```

(e) How many array accesses does a single call to `Y.remove` take in the worst case? (To make this well-defined, we assume that the compiler performs no clever optimisations. That is, every array access we've written in the code will actually be performed.)
A $\sim 4N$
B 2
C $\sim 2N$
D 7

(f) How many array accesses does a single call to the most expensive public method of `Y` take in the worst case?
A linear in k.
B constant.
C linearithmic in k.
D quadratic in k.

(g) What is the number of array accesses per operation in the following sequence of $2k$ operations, starting from an empty data structure: `y.insert(1); y.remove(); y.insert(2); y.remove(); y.insert(3); y.remove(); ... y.insert(k); y.remove();`
A linear in k in the worst case and in the amortized case.
B constant in the worst case.
C constant in the amortized case, but linear in k in the worst case.
D quadratic in k in the worst case.

(h) True or false: The data structure `Y` uses space linear in `N`. Explain you answer on a separate piece of paper. (Be as formal *and short* as you can, but not shorted. If you use more than half a page of text you're on the wrong level of abstraction.)

M.1.5 Modifications in the new version. We modified the exercise by adding some items, so that the prerequisite skills can be assessed separately, that are critical for this question. Namely, we focused on operators (modulus), conditionals, arrays (indexing and storage), and array iteration.

We also add an item focusing on the difference between an object/ADT and its instances, and the difference between values and references. These aspects were not addressed in the original question, but the original code contains a class, thus we expanded the topic a bit.

We kept items from (e) on unmodified, since with the previous addenda, they can be used just to focusing assessment of the advance topics.

The order of items follows this rationale: first the core items that access the advance topics, then items that aim at either confirming that the correct answer is not by chance or by superficial guessing from the code, or to distinguish whether the incorrect answers are due to lack of prerequisite skills or bad knowledge/understanding of advanced concepts.

- The use of "comparable" (lines 24-25) could prevent comprehension of the declaration. This issue was not classified by our code-book since it is language-specific. However, we added a comment before line 24 to make this clear:
  The line below is essentially:
  `Key[] tmp = new Key[2*A.length];`
  with keys being comparable.
  The fact that key are comparable could be also removed, but that would weaken distractor C of item (a).
- A new question is added as (c-1), in order to assess prerequisites on operators (modulus), conditionals, arrays (indexing and storage), and array iteration.
  The correct answer is: `A` holds $\{1, 3, 0, 0, 0, 0, 0, 0\}$, `lo` holds 0, `hi` holds 2, `N` holds 2.
  The second part requires tracing. The instance is not consistent, and this might raise doubts in most conscious students, so we added the idem before, which also assesses their ability to reason about constraints (pre- and post-conditions). A good answer for this item is to notice that `N` is always equal to the distance (in absolute value) between hi and lo, which is not true in this instance. Answering correctly to original items (b) and (c) is a good step towards this understanding. Thus the question addresses circularity, since in this instance `lo` is higher that `hi`.
- We modified question (d) to asses separately the "conditionals" prerequisite skill. Namely we added this question: *How many times was "rebuild" called?*
- Finally, we added another item, (d-1), after (d) to assessing "references" and "instances". The correct answer is `a` holds 1 and `b` holds 2. If they say `a` is 1, they do not know reference semantics; if the say `b` is 3, they do not differentiate instances (they have only *one* queue).

M.2 Concurrency 1

M.2.1 Original question. As a teacher, you are constantly on the hunt for good ideas for exam exercises. The main problem, however, is that it is easy to forget the good ideas before they are actually used to produce a good question. To solve this problem, one teacher implemented a data structure to keep track of them. The implementation of the data structure is below. It has the following operations:

- `idea_init`: Initializes the idea buffer.
- `idea_add`: Adds an idea (a string) to the buffer. If the buffer is full and the idea could not be added, `false` should be returned, otherwise `true` should be returned.
- `idea_get`: Randomly selects and returns an idea from the buffer. The idea is also removed to ensure it is not used for another exam. If no ideas are present, `idea_get` shall wait until a new idea is added with `idea_add`.

During the exam periods, `idea_add` and `idea_get` are used frequently by many teachers. Therefore, it is important that they are usable from multiple threads simultaneously as far as possible.

(1) Is *busy-wait* used somewhere in the implementation? If so, where?
(2) Use suitable synchronization primitives to eliminate any occurrences of *busy-wait* you found.
(3) After using the data structure for a while, some users notice that the same idea has been used multiple times (i.e. multiple calls to `idea_get` returned the same idea). Furthermore, ideas sometimes disappear from the buffer, even though `idea_add` indicates success by returning `true`.
Explain with an example what could have happened when...
  (a) ...the same idea was used multiple times.
  (b) ...the buffer ”lost” one or more ideas.
(4) Mark any critical sections present in the functions `idea_add` and `idea_get`. Also note the resource(s) that need protection.
(5) Use suitable synchronization primitives to synchronize the code based on the critical sections you found.
Note: Strive for a solution that allows maximum theoretical parallellism, even though that solution may perform worse in practice due to synchronization overheads (please note if you think this is the case).
Note: Points may be deducted for excessive locking.

```c
#define BUFFER_SIZE 32

struct idea_buffer {
  // All ideas in the buffer. Empty elements are
  // set to NULL.
  const char *ideas[BUFFER_SIZE];
  // Number of ideas in the buffer.
  int count;
};
// Initialize the buffer.
void idea_init(struct idea_buffer *buffer) {
  for (int i = 0; i < BUFFER_SIZE; i++)
    buffer->ideas[i] = NULL;
  buffer->count = 0;
}
// Add a new idea to an empty location in the
// buffer. Returns 'false' if the buffer is full.
bool idea_add(struct idea_buffer *buffer,
              const char *idea) {
  // Find an empty location.
  int found = BUFFER_SIZE;
  for (int i = 0; i < BUFFER_SIZE; i++) {
    if (buffer->ideas[i] == NULL) {
      found = i;
      break;
    }
  }
  // Full?
  if (found >= BUFFER_SIZE)
    return false;
  // Insert into the buffer.
  buffer->ideas[found] = idea;
  buffer->count++;
  return true;
}
// Get and remove a random element from the
// buffer. If no elements are present, the
// function waits for an element to be added.
const char *idea_get(struct idea_buffer *buffer) {
  while (buffer->count == 0)
    ;
  buffer->count--;
  // Find an element. Start from a random index,
  // and look through the array until we find a
  // non-empty element.
  int pos = rand() % BUFFER_SIZE;
  while (buffer->ideas[pos] == NULL) {
    pos = (pos + 1) % BUFFER_SIZE;
  }
  // Remove it.
  const char *result = buffer->ideas[pos];
  buffer->ideas[pos] = NULL;
  return result;
}
```

M.2.2 Analysis of the original question. This question assesses the following skills categorized in this paper:

- Simple Statements
- Operators
- Assignments
- Tracing
- Debugging
- Loop constructs
- Array iteration
- Types
- Values and references
- Indirection
- Parameters
- Return values
- Function scoping and data flow
- Classes/records/ADT
- Problem decomposition
- Reasoning about constraints
- Meta-tracing knowledge

M.2.3 New version. (Changes highlighted in bold)

As a teacher, you are constantly on the hunt for good ideas for exam exercises. The main problem, however, is that it is easy to forget the good ideas before they are actually used to produce a good question. To solve this problem, one teacher implemented a data structure to keep track of them. The implementation of the data structure is below. It has the following operations:

- `idea_init`: Initializes the idea buffer.
- `idea_add`: Adds an idea (a string) to the buffer. If the buffer is full and the idea could not be added, `false` should be returned, otherwise `true` should be returned.
- `idea_get`: Randomly selects and returns an idea from the buffer. The idea is also removed to ensure it is not used for another exam. If no ideas are present, `idea_get` shall wait until a new idea is added with `idea_add`.

During the exam periods, `idea_add` and `idea_get` are used frequently by many teachers. Therefore, it is important that they are usable from multiple threads simultaneously as far as possible.

(1) **When executing the following code, what is the value of the variable res afterwards?**

```
struct idea_buffer x;
idea_init(&x);
idea_add(&x, "a");
int res = x.count;
```

(2) **When executing the following code, what do you expect the last line to do?**

```
struct idea_buffer a, b;
idea_init(&a);
idea_init(&b);
idea_add(&a, "a");
idea_get(&b); // <-- here?
```

(3) Is *busy-wait* used somewhere in the implementation? If so, where?

(4) Use suitable synchronization primitives to eliminate any occurrences of *busy-wait* you found.

(5) After using the data structure for a while, some users notice that the same idea has been used multiple times (i.e. multiple calls to `idea_get` returned the same idea). Furthermore, ideas sometimes disappear from the buffer, even though `idea_add` indicates success by returning `true`.
Explain with an example what could have happened when...
   (a) ...the same idea was used multiple times.
   (b) ...the buffer ”lost” one or more ideas.

(6) **Mark all lines in the code where some data inside a `idea_buffer` is accessed. Also note which part of the expression that accesses the part.**

(7) **Which variables are not shared between threads?**

(8) Mark any critical sections present in the functions `idea_add` and `idea_get`. Also note the resource(s) that need protection.

(9) Use suitable synchronization primitives to synchronize the code based on the critical sections you found.
Note: Strive for a solution that allows maximum theoretical parallellism, even though that solution may perform worse in practice due to synchronization overheads (please note if you think this is the case).

Note: Points may be deducted for excessive locking.

```
#define BUFFER_SIZE 32

struct idea_buffer {
  // All ideas in the buffer. Empty elements are
  // set to NULL.
  const char *ideas[BUFFER_SIZE];
  // Number of ideas in the buffer.
  int count;
};
// Initialize the buffer.
void idea_init(struct idea_buffer *buffer) {
  for (int i = 0; i < BUFFER_SIZE; i++)
    buffer->ideas[i] = NULL;
  buffer->count = 0;
}
// Add a new idea to an empty location in the
// buffer. Returns 'false' if the buffer is full.
bool idea_add(struct idea_buffer *to,
              const char *idea) {
  // Find an empty location.
  int found = BUFFER_SIZE;
  for (int i = 0; i < BUFFER_SIZE; i++) {
    if (to->ideas[i] == NULL) {
      found = i;
      break;
    }
  }
  // Full?
  if (found >= BUFFER_SIZE)
    return false;
  // Insert into the buffer.
  to->ideas[found] = idea;
  to->count++;
  return true;
}
// Get and remove a random element from the
// buffer. If no elements are present, the
// function waits for an element to be added.
const char *idea_get(struct idea_buffer *from) {
  while (from->count == 0)
    ;
  from->count--;
  // Find an element. Start from a random index,
  // and look through the array until we find a
  // non-empty element.
  int pos = rand() % BUFFER_SIZE;
  while (from->ideas[pos] == NULL) {
    pos = (pos + 1) % BUFFER_SIZE;
  }
  // Remove it.
  const char *result = from->ideas[pos];
  from->ideas[pos] = NULL;
  return result;
}
```

M.2.4 Modifications in the new version. In the new version, we made the following changes:

- The names of the pointer variables were altered to make it impossible to rely entirely on pattern matching in order to arrive at conclusions regarding shared and non-shared variables in parts 1, 2 and 5.

- Part 1 was added, which explicitly assesses that students understand references in C.
- Part 2 was added, which explicitly assesses the *object* category, that the student understands the difference between struct declarations and instances.
- Part 6 was added, which explicitly assesses whether students understand indirection. Since the pointer variables are renamed, students need to be aware that the different variables actually refer to the same instance.
- Part 7 was added, which assesses *function scoping and data flow* by asking the student to note which variables are not shared, which requires the student to understand which variables are local to functions and which are not.

M.3 Concurrency 2

This question presents the student with an implementation of a data structure and asks the student to make sure it is synchronized.

M.3.1 Original question. You are working on a program that is doing heavy computations. Sadly, the program only uses one of the cores in your system, and you got tired of waiting for it to complete all the time. After some thinking, you realized that it is possible to split the problem up into multiple independent parts that can run in parallel most of the time. In order to do this, you implement a basic structure to help you managing the workload. Sadly, something seems to be wrong as you sometimes get zero as a result from many of the parts.

You have implemented two functions: spawn and `wait`:

- spawn creates a thread that executes do_work with the parameter passed to it. "spawn" returns a pointer to `struct work_data` that keeps track of the created thread.
- The pointer returned from spawn may then be passed to `wait` in order to wait for the thread to complete its task and get the result. It should be possible to call spawn from multiple threads concurrently.
  You may assume that `wait` is only called once for each time spawn is called.

Correct any synchronization issues in the implementation.

```c
// Function doing the heavy computations. We
// want to run this in parallel in two threads.
int do_work(int param) {
    // Here we're doing heavy work...

    // Hint, try uncommenting the following
    // line to see the problems occurring
    // more frequently.
    // timer_msleep(param);

    // For simplicity we simply square
    // the parameter.
    return param * param;
}

// Data structure keeping track of a thread
// running "do_work".
struct work_data {
    // Parameter to be passed to "do_work".
    int param;

    // Result from "do_work" in case
    // the thread is done.
    int result;
};

// The first function executed in new threads.
void thread_main(struct work_data *data) {
    data->result = do_work(data->param);
}

// Start a new thread running the function
// "do_work" with "param" as a parameter.
// Returns a "struct work_data" that may be
// passed to "wait" in order to get the result.
struct work_data *spawn(int param) {
    // Allocate a new data structure and
    // initialize it.
    struct work_data *data =
        malloc(sizeof(struct work_data));
    data->param = param;

    // Create a new thread running "thread_main"
    // and give it access to "data".
    thread_new(&thread_main, data);

    return data;
}

// Wait for a thread started with "spawn" to
// complete, and get the result produced. "wait"
// will also free "data", so we assume that "wait"
// is only called once for each call to "spawn".
int wait(struct work_data *data) {
    // Get the result, free the memory
    // and return it.
    int result = data->result;
    free(data);
    return result;
}

// Main function. If the implementation above is
// correct you should not need to change anything
// here. It could be interesting to modify "main"
// in order to test your implementation.
int main(void) {
    struct work_data *a = spawn(10);
    struct work_data *b = spawn(100);

    int c = do_work(5);

    printf("Result for 'a': %d\n", wait(a));
    printf("Result for 'b': %d\n", wait(b));
    printf("Result for 'c': %d\n", c);

    return 0;
}
```

M.3.2 Analysis of the original question. The code implements a simple data structure that acts as a simple version of a *future*, and asks the student to synchronize it. Arriving at a solution requires the student to understand under what conditions the code is assumed to be used, in order to work out that the data in `work_data` needs to be protected, and that `wait` needs to be synchronized appropriately to protect that data. One solution for this exercise is to add a semaphore to the data structure and call *up* on the semaphore at the end of do_work and *down* in the beginning of `wait`.

M.3.3 Summary of assessed skills. This question assesses the following skills categorized in this paper:

- Simple statements
- Assignments
- Types
- Values and references
- Indirection
- Parameters

- Return values
- Function scoping and data flow
- Classes/records/ADT
- Object/instance/variable
- Meta-tracing knowledge

Advanced skills:

- Threads
- Semaphores or condition variables

As the synchronization goal is not explicitly stated, this problem would be in level 3 of the concurrency development levels described previously.

M.3.4 New version. (Changes are highlighted in bold)

You are working on a program that is doing heavy computations. Sadly, the program only uses one of the cores in your system, and you got tired of waiting for it to complete all the time. After some thinking, you realized that it is possible to split the problem up into multiple independent parts that can run in parallel most of the time. In order to do this, you implement a basic structure to help you managing the workload. Sadly, something seems to be wrong as you sometimes get zero as a result from many of the parts.

You have implemented two functions: `spawn` and `wait`:

- `spawn` creates a thread that executes `do_work` with the parameter passed to it. "spawn" returns a pointer to `struct work_data` that keeps track of the created thread.
- The pointer returned from `spawn` may then be passed to `wait` in order to wait for the thread to complete its task and get the result. It should be possible to call `spawn` from multiple threads concurrently.
  You may assume that `wait` is only called once for each time `spawn` is called.

(1) **What do you expect to be printed by the statement on lines 50-51 when running the supplied program?**
(2) **Highlight the lines in the code that access shared data. For each line, highlight the expressions that access shared data.**
(3) **How many instances of `struct work_data` are created when running the main function?**
(4) **Consider the commented line on line 92. What would go wrong if this line was not a comment?**
(5) Use suitable synchronization primitives to synchronize the code.

```
1  // Function doing the heavy computations. We
2  // want to run this in parallel in two threads.
3  int do_work(int param) {
4      // Here we're doing heavy work...
5
6      // Hint, try uncommenting the following
7      // line to see the problems occurring
8      // more frequently.
9      // timer_msleep(param);
10
11     // For simplicity we simply square
12     // the parameter.
13     return param * param;
14 }
15
16 // Data structure keeping track of a thread
17 // running "do_work".
18 struct work_data {
19     // Parameter to be passed to "do_work".
20     int param;
21
22     // Result from "do_work" in case
23     // the thread is done.
24     int result;
25 };
26
27 // The first function executed in new threads.
28 void thread_main(struct work_data *data) {
29     printf("New thread computing %d\n",
30         data->result);
31     data->result = do_work(data->param);
32 }
33
34 // Initialize data.
35 void initialize_data(int param,
36         struct work_data *init) {
37     init->param = param;
38     init->result = 0;
39 }
40
41 // Start a new thread running the function
42 // "do_work" with "param" as a parameter.
43 // Returns a "struct work_data" that may be
44 // passed to "wait" in order to get the result.
45 struct work_data *spawn(int param) {
46     // Allocate a new data structure and
47     // initialize it.
48     struct work_data *data =
49         malloc(sizeof(struct work_data));
50     initialize_data(param, data);
51     printf("Initialized data for %d\n",
52         data->param);
53
54     // Create a new thread running "thread_main"
55     // and give it access to "data".
56     thread_new(&thread_main, data);
57
58     return data;
59 }
60
61 // Version of spawn.
62 struct work_data *spawn2(int param,
63         struct work_data *data) {
64     initialize_data(param, data);
65     printf("Initialized data for %d\n",
66         data->param);
67
68     // Create a new thread running "thread_main"
69     // and give it access to "data".
70     thread_new(&thread_main, data);
71
72     return data;
73 }
74
75 // Wait for a thread started with "spawn" to
76 // complete, and get the result produced. "wait"
77 // will also free "data", so we assume that "wait"
78 // is only called once for each call to "spawn".
```

```c
int wait(struct work_data *wait_for) {
    // Get the result, free the memory
    // and return it.
    int result = wait_for->result;
    free(wait_for);
    return result;
}

// Main function. If the implementation above is
// correct you should not need to change anything
// here. It could be interesting to modify "main"
// in order to test your implementation.
int main(void) {
    struct work_data *a = spawn(10);
    struct work_data *b = spawn(100);
    // b = spawn_2(1000, b);

    int c = do_work(5);

    printf("Result for 'a': %d\n", wait(a));
    printf("Result for 'b': %d\n", wait(b));
    printf("Result for 'c': %d\n", c);

    return 0;
}
```

M.3.5 Modifications in the new version. In the new version we made the following changes:

- Changed the name of the parameter used for `struct work_data` in the functions. By doing this, students need to understand how pointers work in order to find the proper values of the print statements in part 1, and to find shared data in part 2.
- By adding part 2, it also becomes visible if students understand function scope, as they would otherwise indicate local variables as being shared.
- By adding part 3, the student needs to understand the difference between a struct declaration and an instance of that struct. This is also visible by observing the print statement inside `wait`, which is a part of assignment 1.
- The call to `spawn_2` in part 4 also tests the ability to differentiate between a struct declaration and an instance by accidentally re-using one instance for multiple tasks. This prevents guessing the correct number of instances on part 3, but requires understanding of references as well.
- Part 5 is like in the original, and may now be used to verify that the location of the semaphore required by the final solution corresponds to the students' prerequisite skills.

M.4 Advanced OOP: Inheritance and Polymorphism

This question contains a piece of code that defines a number of classes in Eiffel, and asks the student what would happen when a piece of provided code is executed. Interestingly, Eiffel allows for *co-variant* overloading of methods, and still it considers the derived type as conforming to the base one (this is in contrast with the type systems of many popular languages, such as Java and C++). The assessment is designed to make students consider the problems this possibility might cause (since Liskov's conditions do not necessarily hold) in a system in which both the base and the derived component are used; see [52] for the background and the inspiration of the exercise and further discussion.

For readers not familiar with the Eiffel language: member functions and attributes are known as **`feature`**s in Eiffel lingo; **`deferred`** means the implementation is postponed in another type definition, like `abstract` in Java; **`require`**, **`ensure`**, **`invariant`** mark pre-, post-conditions, and invariants; **`create`** mark constructors features and it is needed also to call them; **`Current`** is a self reference; `out` is analogous to `toString` in Java.

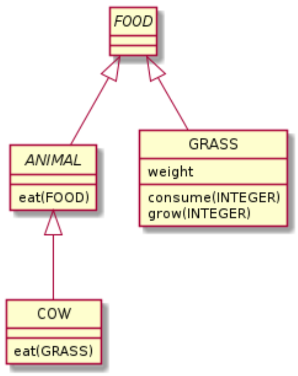


Figure 3: UML Class diagram for assessment classes

M.4.1 Original question. Consider the Eiffel code below and answer the following questions. An UML class diagram is depicted in Figure 3.

(1) consider the assignment `f := g` at line 80 of feature `make` in the class `APPLICATION`. What happens if after that statement one puts a call `c.eat(f)`? Does it cause an error? If yes, explain whether it is a compile-time or a run-time error.
(2) Consider an assignment `f := c`. What happens if after that statement one puts a call `a.eat(f)`? Does it cause an error? If yes, explain whether it is a compile-time or a run-time error.
(3) Consider an assignment `f := a`. What happens if after that statement one puts a call `a.eat(f)`? Does it cause an error? If yes, explain whether it is a compile-time or a run-time error.
(4) Suggest sensible invariants for class `GRASS` and pre-/post-conditions for features `grow` and `consume`.
(5) Suggest a sensible pre-condition for feature `eat` in class `COW` and explain why that would not be effective.
(6) Rewrite `eat` in `COW` such that it raises an exception in case (5). Is it a good idea according to a Design By Contract approach?

```
1  deferred class
2    FOOD
3  end
4
5  deferred class ANIMAL inherit FOOD
6  feature
7
8    eat (f: FOOD)
9      require
10       -- f should not refer to me...
11       no_autophagy: f /= ({FOOD} [Current])
12     deferred
13     end
14
15 end
16
17 class GRASS inherit FOOD
18     redefine
19       out
20     end
21
22 feature
23
24   out: STRING
25     do
26       Result := "a bunch of grass (" + weight.out + "kg)"
27     end
28
29   consume (q: INTEGER)
30     do
31       weight := weight − q
32     end
33
34   grow (q: INTEGER)
35     do
36       weight := weight + q
37     end
38
39   weight: INTEGER
40
41 end
42
43 class COW inherit ANIMAL
44     redefine
45       eat,
46       out
47     end
48
49 feature
50
51   eat (g: GRASS)
52     do
53       g.consume (10)
54     end
55
56   out: STRING
57     do
58       Result := "a cow"
59     end
60
61 end
62
63 class APPLICATION
64 create
65   make
66
67 feature -- Main
68
69   make
70       -- Run application.
71     local
72       a: ANIMAL
73       c: COW
```

```
      g: GRASS
      f: FOOD
    do
      create c
      create g
      a := c
      f := g -- focus on this
      print (a.out + " is going to eat: " + f.out + "%N")
      a.eat (f)
    end

end
```

M.4.2 Answers to the original question.

(1) The formal parameter (a static, compile-time property) of `c.eat` is a `GRASS` and an actual parameter (a dynamic, run-time property) of `FOOD` (the static type of `f`) is not compatible with it, it raises a compile-time error.

(2) From a static, compile-time viewpoint the statement `a.eat(f)` is fine, since the type of the formal parameter and the static type of `f` are compatible (`FOOD` in both cases). At run-time, however, a **change of availability of type** ("catcall") error is caught, since dynamically `a` is a `COW` and it expects a `GRASS` to `eat`, but `f` is a `COW`.

(3) From a static, compile-time viewpoint the statement `a.eat(f)` is fine, since the type of the formal parameter and the static type of `f` are compatible (`FOOD` in both cases). At run-time, however, a **change of availability of type** ("catcall") error is caught, since dynamically `a` is a `COW` and it expects a `GRASS` to `eat`, but `f` is a `COW` (the dynamic type of `a`).

(4) Invariants and pre/post-conditions that make sense:

```
class GRASS inherit FOOD -- only the relevant code
      is reported here

feature
  consume (q: INTEGER)
    require
      q > 0
      weight >= q
    do
      weight := weight - q
    ensure
      weight = old weight - q
    end

  grow (q: INTEGER)
    require
      q > 0
    do
      weight := weight + q
    ensure
      weight = old weight + q
    end

invariant
  weight >= 0

end
```

(5) Having a pre-condition on `COW.eat` on `g.weight` would be perfectly sensible, for example `g.weight >= 10`. However, the syntactical enforcement of Liskov's substitution principle embedded in Eiffel would put this condition in **`or else`** (**`require else`**) with the pre-condition of the base class `FOOD` (i.e., `True`), therefore this check will be ineffective at run-time (but could be still useful as a hint to the user of the class `COW`).

```
class COW inherit ANIMAL -- only the relevant code
      is reported here
    redefine
      eat
    end

feature

  eat (g: GRASS)
    require else
      g.weight >= 10
    do
      g.consume (10)
    end

end
```

(6) One could add a check:

```
class COW inherit ANIMAL -- only the relevant code
      is reported here
    redefine
      eat
    end

feature

  eat (g: GRASS)
    do
      check g >= 10
      g.consume (10)
    end

end
```

This is not a good idea according to Design By Contract, since the constraint is not available to the clients of `COW`. However, the constraint is probably part of the contract of `GRASS.consume` (see answer (4)), thus a better idea would be to avoid the contract violation by growing some `GRASS`.

```
class COW inherit ANIMAL -- only the relevant code
      is reported here
    redefine
      eat
    end

feature

  eat (g: GRASS)
    do
        if g.weight < 10 then
          g.grow (10 - g.weight)
        end
        g.consume (10)
    end

end
```

M.4.3 Summary of assessed skills. Although the goal of the exercise is to test the understanding of the co-variant overloading and its relationship with static typing constraints and Liskov's substitution principle (in a correct system, a component C' can be substituted to component C only if the pre-conditions for the use of C' are weaker or equal than the pre-conditions of C, and the post-conditions of C' are stronger or equal than the post-conditions of C), the answering student must master at least the following fundamental skills (from our code-book in Section 4).

- Simple Statements
- Operators
- Assignments
- Tracing

- Types
- Values and references
- Parameters
- Classes/records/ADT
- Meta-tracing knowledge

If some of these prerequisites are not clear, wrong answers cannot be clearly attributed to misconceptions in the advanced topics. In particular, a familiarity with a tracing as a general strategy to understand how the interpreter executes the code is needed, but this strategy has to transferred on a new level, since it should applied not only on (abstract) states, but also to types and method dispatching.

M.4.4 New version. To address some of these issues, the following changes could be made:

1. Add a new concrete `FOOD` class (e.g., `PLANKTON`) and create an object `p` from this class;
2. add a method `log_food` with a parameter `x` of type `FOOD`, the method just prints the dynamic type of the actual parameter bound to `x`;
3. add a question about the output of `log_food(p)`, `log_food(g)`, `log_food(f)`, where `g` is a reference to a `GRASS` object and `f` is a reference to a `GRASS` object of `FOOD` static type (using `f` as an actual parameter of `log_food` before assigning it to a concrete object is not legal in Eiffel, unless the type is marked explicitly as `detachable`, see `log_food2`);

```
-- ANIMAL, COW, GRASS, and FOOD as before

class PLANKTON inherit FOOD -- new class
  redefine
    out
  end

  feature

    out: STRING
    do
      Result := "Lots of planksters"
    end
end

class APPLICATION -- modified

  feature -- Main

    make
        -- Run application.
      local
        a: ANIMAL
        c: COW
        g: GRASS
        f: FOOD
        p: PLANKTON
      do
        create c
        create g
        create p
        g.grow (5)
        a := c

            -- log_food2(f) -- log_food(f) not legal
        f := g

        log_food(p)
        log_food(g)
        log_food(f)

        print (a.out + " is going to eat: " + f.out + "%N")
        a.eat (f)
        print ("Finished!%N")
      end

    log_food(x: FOOD)
    do
      print ("The food x is: " + x.out + "%N")
    end

    log_food2(x: detachable FOOD)
    do
      if attached x then
          print ("The food x is: " + x.out + "%N")
      else
        print("No x%N")

      end
    end

end
```

M.5 BDSI: B.6

We modified some questions from the Basic Data Structures Inventory (BDSI) [66], for the purposes of exploring potential changes to question designs, which might also explicitly assess prerequisite skills. However, we have not done any empirical validity experiments on these modified versions yet, to see how actual learners respond to them - for example, having learners think aloud as they answer the question. Thus, we do not really know how good they are yet. If you want to maintain the validity argument for the BDSI (especially if you are using the BDSI for summative use, like to assess learning at the end of a class), do not substitute or add these modified questions, and do not use them as practice or for class assignments, as they are too close to the BDSI questions.

With that said, here is our modified question based on question 6 from the BDSI.

M.5.1 Original question. Here is a possible method for a `SinglyLinkedList` class. Assume `head` is a variable in the `SinglyLinkedList` class that refers to the first node in the list.

```
DEFINE mystery(value)
 current = head
 temp = nil
 WHILE current != nil AND current.item != value DO
  temp = current
  current = current.next
 ENDWHILE
 RETURN temp
ENDDEF
```

Which of the following best explains the "purpose" of the `mystery` method? (That is, what is the overall goal of the `mystery` method?)

Select one of the following statements:

A It returns the node before the one containing `value`, or it returns `nil` if `value` is in the head, or it returns the last node of the list of `value` if not found.
B It returns the node containing `value`, or it returns the last node of the list if `value` is not found.
C It returns the node containing `value`, or it returns `nil` if `value` is not found.
D It returns the node before the one containing `value`, or it returns `nil` if `value` is in the head, or it returns `nil` if `value` is not found.

M.5.2 Analysis of the original question. This question assesses the following prerequisite skills:

- Operators
- Assignments
- Loop constructs
- Indirection
- References
- Meta-tracing knowledge

M.5.3 New version. (Changes are highlighted in bold)

Here is a possible method for a `SinglyLinkedList` class. Assume `head` is a variable in the `SinglyLinkedList` class that refers to the first node in the list.

```
DEFINE mystery(value)
 current = head
 temp = nil
 WHILE current != nil AND current.item != value DO
  temp = current
  current = current.next
 ENDWHILE
 RETURN temp
ENDDEF
```

Which of the following statements **is correct/matches the "purpose"** of the `mystery` method? (That is, matches the overall goal of the `mystery` method?)

Select all that apply:

A **`mystery` returns the node containing `value`, or it returns the last node of the list of `value` is not in the list.**
B **For the `SinglyLinkedList` holding 2, then 4, then 6, `mystery(1)` will return the last node in the list.**
C **If the value is not in the list, `mystery` returns `nil`.**
D **If the value is in the list, `mystery` returns the node containing the value.**
E **If the value is in the list, `mystery` returns the value contained in the node in the list.**
F **If the value is in the head node of the list, `mystery` returns nil.**

M.5.4 Modifications in the new version. The differences between the original question and the new version is that all answer alternatives have been replaced, and the student is allowed to select multiple correct answers. The correct answer is selecting B and F, which implies good meta-tracing skills and knowing other prerequisites for this question.

M.6 BDSI: B.8

We modified a question from the Basic Data Structures Inventory (BDSI) [66], for the purposes of exploring potential changes to question designs, which would add additional required knowledge to the question. However, we have not done any empirical validity experiments on these modified versions yet, to see how actual learners respond to them - for example, having learners think aloud as they answer the question. Thus, we do not really know how good they are yet. If you want to maintain the validity argument for the BDSI (especially if you are using the BDSI for summative use, like to assess learning at the end of a class), do not substitute or add these modified questions, and do not use them as practice or for class assignments, as they are too close to the BDSI questions.

With that said, here is our modified question based on question 8 of the BDSI.

M.6.1 Original question. Suppose that your program stores a list of Strings. The user is permitted to access the string at a given position (index) in the list, and can make as many accesses as they wish. `N` is the number of strings in the list.

Which `List` data structure would provide the best performance for the user accesses and why?

Select one:

A `ArrayList` is best as it guarantees constant-time access.
B `ArrayList` is best as it guarantees access time proportional to log N using binary search.
C Unsorted `DoublyLinkedList` is best as it guarantees constant-time access.
D Unsorted `DoublyLinkedList` is best as it guarantees access time proportional to log N using binary search.
E Sorted `DoublyLinkedList` is best as it guarantees constant-time access.
F Sorted `DoublyLinkedList` is best as it guarantees access time proportional to log N using binary search.

M.6.2 Analysis of the original question. This question is interesting, as it does not assess any of the prerequisite skills, it only assesses the students' knowledge of containers and their properties.

M.6.3 New version. (Changes are highlighted in bold)

Suppose that your program stores a list of Strings **in a variable called `the_list`**. The user is permitted to access the string **like `the_list.get(x)`, where `x < N` and** `N` is the number of strings in the list.

Which `List` data structure would provide the best performance for the user accesses and why?

Select one:

A `ArrayList` is best as it guarantees constant-time access.
B `ArrayList` is best as it guarantees access time proportional to log N using binary search.
C Unsorted `DoublyLinkedList` is best as it guarantees constant-time access.
D Unsorted `DoublyLinkedList` is best as it guarantees access time proportional to log N using binary search.
E Sorted `DoublyLinkedList` is best as it guarantees constant-time access.
F Sorted `DoublyLinkedList` is best as it guarantees access time proportional to log N using binary search.

M.6.4 Modifications in the new version. We replaced the English description of random access by position with the code for that, to also assess understanding of array syntax/semantics. This is an example of how one might add some more prerequisite skills to a question, by removing natural language descriptions and replacing them with notation.

O OTHER QUESTIONS

O.1 Scientific Computing

The following question is given as a pre-exam to non-CS students taking a course on Scientific Computing in Python. The idea is to ensure that students are able to write and run Python code on their computers, and to assess basic programming ability:

- Write a script to compute the numeric integral of $\cos(x)$ from 0 to $\pi/2$.
- Use the "left rectangles" approach and $N = 1000$ intervals.
- Hint: use a for loop. Add up the areas of all the slices.

O.2 Software Design Question 1

The following question was given on a midterm exam on a question in a course on software design methods:

Recall the Pharmacy and PharmacyDB classes from the project:

```java
public class Pharmacy {
    private String id;
    private String owner;
    private String busName;
    private String address;
    private String suite;
    private String city;
    private String state;
    private String zip;
    private String phone;
    private String type;

    public Pharmacy() {}

    public Pharmacy(String id, String owner,
        String busName, String address, String
        suite, String city, String state, String
        zip, String phone, String type) {
        this.id = id;
        this.owner = owner;
        this.busName = busName;
        this.address = address;
        this.suite = suite;
        this.city = city;
        this.state = state;
        this.zip = zip;
        this.phone = phone;
        this.type = type;
    }

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public String getOwner() {
        return owner;
    }

    public void setOwner(String owner) {
        this.owner = owner;
    }

// getters and setters for many instance variables
    are not shown to save space
    public String getZip() {
        return zip;
    }

    public void setZip(String zip) {
        this.zip = zip;
    }

    public String getPhone() {
        return phone;
    }

    public void setPhone(String phone) {
        this.phone = phone;
    }
}
```

This is a shortened version of PharmacyDB:

```java
public class PharmacyDB {
    private HashMap<String, Pharmacy> pharmMap =
        new HashMap<String, Pharmacy>();
    public void add(Pharmacy pharm)  {
        pharmMap.putIfAbsent(pharm.getId(), pharm)
            ;
    }
    public Pharmacy getPharmById(String id) {
       return pharmMap.get(id);
    }
    public Boolean containsId(String id) {
       return pharmMap.containsKey(id);
    }

    /**
     * return a list of pharmacies sorted by zip
         code
     */
    public List<Pharmacy> getPharmaciesSortedByZip
        () {
        // needs to be implemented
    }
}
```

(a) Here is a JUnit test class for PharmacityDB. Write a test method for getPharmaciesSortedByZip, which returns an array list of pharmacy objects sorted by zip code. You can just test that the zip codes are in the expected order, rather than testing all the pharmacy values.

```java
class PharmacyDBTest {
    private PharmacyDB pharmDB;
    private Pharmacy pharm1;
    private Pharmacy pharm2;
    private Pharmacy pharm3;

    @BeforeEach
    void setUp() throws Exception {
        pharmDB = new PharmacyDB();
        pharm1=new Pharmacy("1","owner1","
            CVS1","addr1","", "city1","
            state1", "10709", "111-1111","
            pharmacy");
        pharm2=new Pharmacy("2","owner2","
            CVS2","addr2","22", "city2","
            state2", "22222", "222-2222","
            pharmacy");
        pharm3=new Pharmacy("3","owner3","
            CVS3","addr3","3", "city3","
            state3", "333333", "333-3333","
            pharmacy");
        pharmDB.add(pharm2);
        pharmDB.add(pharm1);
        pharmDB.add(pharm3);
    }
}
```

(b) Now write the implementation of the getPharmaciesSortedByZip method. It should return an array list of pharmacy objects sorted by zip code.

O.3 Software Design Question 2

The following question was given on a final exam on a question in a course on software design methods:

Here is an alternative version of the product list. This one uses a low level array to maintain the list of products. It has two methods. One method adds a product, returning false if there is no more space in the array and true otherwise. The other returns the product at a given position in the array. If there is no product at the given position, it returns null.

```
public class ProductList {
    private final int LEN = 3;
    private Product[] products = new Product[LEN];
    private int numProds = 0;

    /**
     * add a product if there is room
     * @param prod
     * @return true if the product can be added,
     * false if there is no more space
     */
    public Boolean add(Product prod) {
        if (numProds >= LEN) {
            return false;
        }
        products[numProds] = prod;
        numProds++;
        return true;
    }

    /**
     * return the product at position pos
     * if there is no product at that position,
     * return null
     * @param pos
     * @return a product or null if no product
     * at that position
     */
    public Product getAtPos(int pos) {
        if (pos >= numProds) {
            return null;
        }
        return products[pos];
    }
}
```

(a) Write a JUnit test class with test methods for the getAtPoint method (don't worry about the add method). You need to test for both possible return values.

(b) You can't use the built in iterator class with low level arrays, so you must write your own. Write the iterator implementation for ProductList as well as the method that returns the iterator. The iterator should implement this interface.

```
public interface MyIterator {
    public Product next();
    public boolean hasNext();
}
```

Here is a program that uses the iterator.

```
public class ProdFun {
    public static void main(String[] args) {
        ProductList products = new
            ProductList();
        products.add(new Product("2A", "
            Friskies Fishalicious Cat Food",
             12.99));
        products.add(new Product("1B", "Fancy
             Feast Cat Food", 11.88));
        products.add(new Product("1C", "
            Friskies Surf N Turf Cat Food",
            10.99));

        MyIterator iter = products.
            getIterator();
        while (iter.hasNext()) {
            Product prod = iter.next();
            System.out.println(prod.getName()
                );
        }
    }
}
```

When run, it prints out:

```
Friskies Fishalicious Cat Food
Fancy Feast Cat Food
Friskies Surf N Turf Cat Food
```

Implement the iterator class, which will be a nested class inside ProductList, as well as the getIterator method.

O.4 Advanced Data Structures and Algorithms

This question appeared as a part of a larger exercise in a final exam for a course on advanced data structures and algorithms. The test contained a number of questions, each stating that one of the attached problems could be solved using some well-known algorithm. Below is the question related to one of the problems:

One of the problems in this set is easily solved by a reduction to network flow.

(a) Which one?

(b) Explain the reduction. Start by drawing the graph corresponding to Sample Input 1. Be ridiculously precise about which nodes and arcs there are, how many there are (in terms of size measures of the original problem), how the nodes are connected and directed, and what the capacities are. Describe the reduction in general (use words like “every node corresponding to a giraffe is connected to every node corresponding to a letter by an undirected arc of capacity the length of the neck”). What does a maximum flow mean in terms of the original problem, and what size does it have in terms of the original parameters?

(c) State the running time of the resulting algorithm, be precise about which flow algorithm you use. (Use words like “Using Bellman-Ford, the total running time will be $O(r^{17}\log^3\epsilon + \log^2 k$ where r is the number of froontzes and k denotes the maximal weight of a giraffe.”). This part merely has to be correct. There is no requirement about choosing the cleverest flow algorithm.

The associated problem statement:

Messy Arithmetic

You’ve finally found your old maths homework from school! Unfortunately, your handwriting when you were a kid was even worse than it is today, and you can’t make out the difference between +, - and *. Also, the exercises and solutions are on separate sheets of paper, and you don’t know which exercise corresponds to which solution.

You want to bring these important historical documents back in shape, in case your future biographer needs them when you’re famous. Certainly there will be no time for this kind of busywork once you are a famous YouTuber or have won two Nobel prizes in a year.

Match each exercise, which is just a pair of numbers with an unreadable arithmetic operation between them, to a solution, which is also just a number. This requires you to determine the proper arithmetic operation between each pair of numbers.

To avoid having to think about rounding errors, let’s assume all numbers are integers. (This is also why we exclude difision from this exercise.)

Input

The first line of input consists of the integer n, the number of exercises. The next line contains the solutions $s_1, \ldots, s_n$ as n integers separated by space. Then follow n lines each containing a pair of integers $a_i b_i$ for $i \in \{1, \ldots, n\}$, separated by space.

Output

The output consists of n lines of the form $a_i \mathrm{op}_i b_i = s'_i$ for $i \in \{1, \ldots, n\}$. The values a_i and b_i are given in the input, and in the same order. The operator op_i is one of +, -, *. The set of values $\{s'_1, \ldots, s'_n\}$ is the same set as $\{s_1, \ldots, s_n\}$, but may be in a different order than given in the input.

You can assume that a solution exists. If there is more than one solution, any one of them will do.

Sample Input 1	**Sample Output 1**

`5`
`0 1 2 3 9`
`1 2`
`1 -1`
`0 5`
`3 3`
`5 4`
	`1 + 2 = 3`
	`1 - -1 = 2`
	`0 * 5 = 0`
	`3 * 3 = 9`
	`5 - 4 = 1`

Sample Input 2	**Sample Output 2**

`4`
`3 2 2 3`
`3 1`
`1 1`
`3 1`
`3 1`
	`3 - 1 = 2`
	`1 + 1 = 2`
	`3 * 1 = 3`
	`3 * 1 = 3`

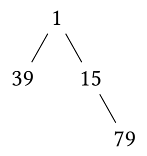
O.5 Data Structures and Algorithms 1

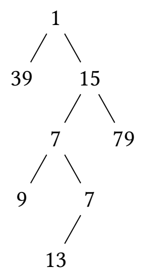
This is an exercise in the final lab exam of a second year course on “data structures and algorithms”. The language taught in the course is C, so the exam needs to assess also knowledge of C and ability to use it to manipulate data structures:

Consider the below, where the type `Bit_node` is used to implement the nodes in a binary tree:

```c
struct bit_node {
  int item;
  struct bit_node *l, *r;
};

typedef struct bit_node *Bit_node;

void printArray( int *a, int n ) {
  for ( int i = 0; i < n; i++ )
    printf( "%d ", a[i] );
  printf( "\n" );
}

void f_r( Bit_node root, int *path, int len ) {
  if (root == NULL )
    return;

  if ( root -> item % 2 ) {
    path[len] = root -> item;
    len++;
  }

  if ( root -> r == NULL && root -> l == NULL ) {
    printArray( path, len );
    return;
  }

  f_r( root -> l, path, len );
  f_r( root -> r, path, len );
}

void f( Bit_node root ) {
  Item *path = malloc( 1000 * sizeof( int ) );
  f_r( root, path, 0 );
}
```

Consider the two binary trees below when answering the following questions:





(1) What height does the stack reach if the `f` function is invoked on the root of the left tree?

(2) What height does the stack reach if the `f` function is invoked on the root of the right tree?

(3) In general, how many lines does the function print if invoked on the root of any binary tree?

(4) What does the function print if invoked on the root of any binary tree that contains only even numbers?

(5) Complete the following phrase: If `root` is the pointer to the root of a binary tree, then the invocation of the function `f1(root)` outputs...

O.6 Data Structures and Algorithms 2

The following question is used as a question on the final examination in a course on Data Structures and Algorithms in order to reveal potential misconceptions regarding basic programming skills:

In the following, you can see two algorithms computing the power function (x^n) for integers x and n. Read through all the questions below without answering them and after that *familiarize yourself with the code thoroughly*. After this, answer all the questions and take time to ponder and explain your reasoning. Note, however, that all the questions refer to the given algorithms. In addition, the *argumentation* is the only thing that matters for the points!

```
1  Algorithm pow1(x, n)
2      if (n = 0)
3          return 1; else
4      if (n = 1)
5          return x; else
6      if ("n is odd")
7          return pow1(x*x, n/2)*x; else
8      if ("n is even")
9          return pow1(x*x, n/2);
10
11 Algorithm pow2(x, n)
12     p = 1;
13     i = 1;
14     while (i <= n)
15         p = p * x;
16         i = i + 1;
17     return p;
```

(a) *Describe* in your own words how `pow1` works (without an example). Note! Try to explain how the algorithm behaves in general. Do not explain the algorithm line by line.

(b) *Describe* in you own words how `pow2` works (without an example). How is it different from the previous one?

(c) In which code line is the **first** multiplication performed? What are the factors (*multiplicand* and *multiplier*) in this case?

(d) In which code line is the **last** multiplication performed? What are the factors (*multiplicand* and *multiplier*) in this case?

(e) *Which lines of code* and *how many multiplications* are totally executed by `pow1`? *Give an example* of the execution of `pow1(2, 9)`.

(f) *Analyze* the time complexity of Algorithm 1 in terms of the input size n.

(g) *Analyze* the time complexity of Algorithm 2 in terms of the input size n.

(h) *Analyze* the time complexity of Algorithm 1 if the line 7 was changed to `return x * pow1(x*x, n/2); else`. *Give an example.*

(i) Is it possible to replace the *while*-loop in Algorithm 2 with another loop construct? Either argue why not or give an example of how to replace it (rewrite the algorithm).

O.7 Data Structures and Algorithms 3

The following question is used as a question on the final exam in a course on data structures and algorithms in order to reveal potential misconceptions regarding basic programming skills:

Below you can see two algorithms, `linearSearch` and `binarySearch`.

```
def linearSearch(table, x):
    for i in range(len(table)):
        if (table[i] == x):
            return i
    return i

def binarySearch(table, x):
    low = 0
    high = len(table) - 1

    while (low <= high):
        mid = (low + high) // 2
        print(low, mid, high)
        if (table[mid] < x):
            low = mid + 1
        elif (table[mid] > x):
            high = mid - 1
        else:
            return mid
    return -1
```

(a) Which of the following statements are true for `linearSearch` (L) and/or `binarySearch` (B) in case of successful search? Use one of the following five options in each case:
L = the statement is correct *only* for `linearSearch`
B = the statement is correct *only* for `binarySearch`
L&B = the statement is correct for *both* `linearSearch` and `binarySearch`
neither = not either or,
I don't know.
Each statement is worth two points as follows: correct answer +2, incorrect answer -1, empty or **I don't know** 0 points. However, you will get at least 0 points from all the questions, and the maximum is 6 x 2 p = 12 points.
  i The algorithm goes through the items from smallest index to the largest.
  ii The algorithm returns always the smallest index for the item x.
  iii The algorithm returns all the indices that have the item x.
  iv The algorithm always goes through all the items.
  v The algorithm is correct only if the array is sorted in ascending order.
  vi The algorithm always returns -1 at the end.

(b) *Give and example to* `linearSearch` the item `x = 14` from the table below. List all the values the variable i holds during the search.

(c) *Give an example to* `binarySearch` the item `x = 14` from the table below. List the values for the variables each time `print(low, mid, high)` is called.

(d) *Argue* whether the following statement is true or false: `linearSearch` is a more efficient algorithm than `binarySearch` to find a single item from an array. Hint: try to justify both alternatives!

table	-9	-1	0	13	14	14	27	29	31	34	36	36	44	44	98
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14