Università degli studi di Udine

Developing Assessments to Determine Mastery of Programming Fundamentals

(Article begins on next page)

31 July 2024

Developing Assessments to Determine Mastery of Programming Fundamentals

Andrew Luxton-Reilly
University of Auckland
New Zealand
andrew@cs.auckland.ac.nz

Brett A. Becker
University College Dublin
Ireland
brett.becker@ucd.ie

Yingjun Cao
University of California, San Diego
USA
yic242@eng.ucsd.edu

Roger McDermott
Robert Gordon University
Scotland, UK
roger.mcdermott@rgu.ac.uk

Claudio Mirolo
University of Udine
Italy
claudio.mirolo@uniud.it

Andreas Mühling
Kiel University
Germany
andreas.muehling@informatik.
uni-kiel.de

Andrew Petersen
University of Toronto Mississauga
Canada
andrew.petersen@utoronto.ca

Kate Sanders
Rhode Island College
USA
ksanders@ric.edu

Simon
University of Newcastle
Australia
simon@newcastle.edu.au

Jacqueline Whalley
Auckland University of Technology
New Zealand
jacqueline.whalley@aut.ac.nz

ABSTRACT

Current CS1 learning outcomes are relatively general, specifying tasks such as designing, implementing, testing and debugging programs that use some fundamental programming constructs. These outcomes impact what we teach, our expectations, and our assessments. Although prior work has demonstrated the utility of single concept assessments, most assessments used in formal examinations combine numerous heterogeneous concepts, resulting in complex and difficult tasks. As a consequence, teachers may not be able to diagnose the actual difficulties faced by students and students are not provided with accurate feedback about their achievements. Such limitations on the nature and quality of feedback to teachers and students alike may contribute to the perceived difficulty and high dropout rates commonly observed in introductory programming courses.

In this paper we review the concepts that CS education researchers have identified as important for novice programming. We survey learning outcomes for introductory programming courses that characterize the expectations of CS1 courses, and analyse assessments designed for CS1 to determine the individual components of syntax and semantics required to complete them. Having recognized the implicit and explicit expectations of novice programming courses, we look at the relationships between components and progression between concepts. Finally, we demonstrate how some complex assessments can be decomposed into atomic elements that can be assessed independently.

KEYWORDS

ITiCSE working group; CS1; learning outcomes; assessment; exam; questions; mastery; learning; computer science education; novice programming

1 INTRODUCTION

A substantial body of research provides evidence that students struggle to pass the program comprehension and program generation tasks that we set [38, 47, 89, 91]. The failure of students to correctly complete these tasks has led many academics to conclude that learning to program is inherently difficult [6, 62]. However, it has recently been argued that the difficulty is due to the assessment tasks that we use, rather than the subject itself [41, 46].

Typical introductory programming courses (which we will, for convenience, call "CS1 courses" using the northern American descriptor) frequently require students to complete assessment tasks that are complex and involve many heterogeneous concepts within a single question. The Computing Curricula 2013 [4] lists a number of learning outcomes for software development fundamentals, and they are typically complex. For example:

(1) Design, implement, test, and debug a program that uses each of the following fundamental programming constructs: basic computation, simple I/O, standard conditional and iterative structures, the definition of functions, and parameter passing. [Usage]
(2) Write a program that uses file I/O to provide persistence across multiple executions. [Usage]

The study by McCracken et al. [47] is a case in point. In this study, students were asked to write a simple calculator that neatly

fits the requirements of Learning Objective (1) above: students were required to design, implement, and test a program (the calculator) that involved writing functions, loops, if statements, and input/output. Programs were collected from 216 students in four different universities. The students scored very poorly – 22.89 out of 110 possible points on average. As a result, the paper has been widely cited since its publication in 2001 for the proposition that students cannot write code (see, e.g., [24, 29, 38, 75]).

A 2013 replication of this experiment, however, found room for partial credit at a more "atomic" level. Only eight students (out of 40) turned in a complete solution. On the other hand, "many solutions were incomplete, but generally the features that were implemented worked" [46, p. 94].

Similar results have been found in a teaching context. A study of 20 introductory exams found that, overall, more than half the marks (54%) were allocated to questions involving between six and 24 lines of code [70]. Another study of nine exams from three universities found that the exam questions required an average of 10.85 different syntax elements [42]. Students who struggled with any of the syntax elements in a given question would have difficulty correctly answering that question. Furthermore, although teachers have a great deal of experience setting exam questions, they regularly underestimated the number of concepts required to complete exam questions [57]. This dense intertwining of concepts is thought to be one of the main reasons that students struggle to succeed in programming courses [61].

The exclusive use of these traditional assessments results in two major problems. First, teachers who are administering the assessments find it difficult to quantify what students are struggling with. A program that fails to compile or fails to pass test cases provides little information about what a student can successfully achieve. For example, an analysis of a large set of solutions submitted by students indicated that both conditionals and loops were problematic for students, with the majority of students failing to submit a correct solution on their first attempt. However, further analysis of the code revealed that more than 50% of students attempting to solve a simple problem using a loop failed to correctly answer the question because they used integer division rather than floating point division [15]. Although the problem was designed to assess the loop construct, students failed due to a misunderstanding of the type resulting from an arithmetic operation.

Second, the exclusive use of traditional assessments deprives the students of feedback about what they do know and what they can achieve. In a meta-study of feedback on student learning, Hattie and Timperley [30] report that feedback is one of the most significant factors that influence student learning. When assessment tasks are designed to build progressively and involve many interrelated concepts, fragile knowledge in one area makes it difficult to demonstrate success in other areas, potentially enhancing feelings of falling behind. Students drop courses when they feel that they are starting to fall behind [57], yet we know that students with fragile knowledge of programming concepts have the potential to succeed if they persist in their studies [86].

Previous work by Zingaro et al. [94] demonstrated that it was possible to effectively assess solitary programming concepts. Further, they assert that:

> *…questions targeted to each skill are more indicative of specific abilities than questions testing these skills in tandem* [94, p. 258].

We speculate that focusing on independent components of programming may increase opportunities for novices to demonstrate what they can achieve, and may improve diagnosis of student difficulties. In this report, we identify what components of programming we expect students to learn in CS1 courses, how we currently assess them, and how we might assess them independently.

1.1 Research Questions

In order to determine what we should be assessing in CS1, we first need to establish what content is covered by CS1 courses and what we expect from students.

We determine these expectations by looking at three different sources of data: the concepts that are reported as being important in the CS Education literature; the explicitly stated learning outcomes for introductory programming courses; and the implicit expectations that can be determined by analysis of the assessments designed for use in introductory programming courses. We also need to know how these specific elements relate to one another so that we can determine which elements might be assessed independently. Finally, we need to demonstrate how we might decompose traditional assessments into independent parts. This leads to the following research questions:

RQ 1 What are the syntax and semantics involved in introductory programming courses?

　RQ 1.1 What syntax and semantics are considered to be important by CS researchers?

　RQ 1.2 What syntax and semantics appear in learning outcomes?

　RQ 1.3 What are the elements of syntax and semantics that we assess?

RQ 2 How are the elements of syntax and semantics related to each other?

RQ 3 How might we assess elements independently?

1.2 Organisation of this Paper

The remainder of this paper is organised as follows. Section 2 reviews related work in five subsections: educational theory, pedagogy, and three subsections on work pertaining to each of our three main research questions. Following this, Sections 3, 4, and 5 focus respectively on research questions 1.1 (concepts from the literature), 1.2 (learning outcomes), and 1.3 (assessment). Section 6 then explores research question 2, and Section 7 explores research question 3. Section 8 provides a discussion of these results before we present our conclusions and future work in Sections 9 and 10.

2 RELATED WORK

2.1 Educational Theory

Constructivist theory claims that each student builds their own understanding through their individual interactions with the world [5]. Students typically come to computing with at least partial understanding of some computational concepts [77]. However, building correct mental models is challenging for novices [79]. Breaking

tasks down into smaller parts may assist them to more easily build understanding by reducing the *cognitive load* imposed by more complex tasks.

Cognitive load is the amount of mental effort required by a given task [82]. When the cognitive load of a given task exceeds the capacity of working memory, learning can be impeded. The cognitive load imposed by a task can be divided into three categories: intrinsic, extraneous, and germane.

intrinsic: Intrinsic cognitive load is the mental effort required by the nature of the task. It is a combination of the difficulty of the task and characteristics of a learner. Different tasks involve different intrinsic cognitive load, but this load cannot be reduced for a given task.

extraneous: This mental effort can be affected by the instructional design of the task. For example, the use of worked examples has been shown to reduce extraneous cognitive load [78].

germane: Like extraneous cognitive load, germane cognitive load is affected by the instructional design of the task, but differs from extraneous in that the extra mental effort is effective for learning.

Germane cognitive load might be considered to be effort rather than load: "one might say that intrinsic and extraneous cognitive load concern cognitive activities that must unavoidably be performed, so they fall under cognitive load; germane cognitive load is the space that is left over that the learner can decide how to use, so this can be labelled as cognitive effort" [21]. With this distinction, the traditional educational goal of reducing cognitive load makes more sense.

Cognitive load can be reduced by the use of schemas. A schema is a framework, or pattern, that can be used to meaningfully understand the world. Schemas are used to organise information so it can be effectively stored and retrieved. Piaget [59] argued that information is stored in long-term memory by processes of *assimilation* and *accommodation*. Assimilation involves activating an existing schema, whereas accommodation involves adaptation of an existing schema or creation of a new schema. Having developed schemas for common tasks, experts can easily assimilate new information and are not impeded by its cognitive load [83].

Novices, however, use much of their working memory to process information because the schemas they have developed are unable to assimilate the information required to complete the task. This is believed to be particularly true of programming, since it is a complex domain that involves many different components and relationships between the components [61, 84].

Although some tasks are too difficult for novices to complete holistically, it may be possible to divide these tasks into smaller components (sub-tasks) that are within the novice's mental capacity to complete. It is possible that the schema required to perform a given task may be broken up into sub-schemas to solve sub-tasks [50, 68]. Once acquired, these schemas may be recombined to solve a task with high intrinsic cognitive load without unduly taxing the learner.

2.2 Pedagogy

Two pedagogical approaches that are generally consistent with our breakdown of basic programming into atomic elements are Mastery Learning, which has been applied in various CS contexts, and the Keller approach, which has its roots in teaching introductory psychology.

2.2.1 Mastery Learning.
Mastery learning [8, 9], or "learning for mastery," divides content to be learned into short units of study which are assessed at the end of the unit. Students are only permitted to continue when they demonstrate mastery of the content they have covered in the unit. Students who fail to achieve competency in the tests are provided with additional help, frequently in the form of individual or group tutorials, before resitting the test [34].

In traditional models of teaching, content is delivered in a fixed time period (such as a semester), and students reach varying levels of competency within the fixed time. Bloom believed that almost all students could reach an expected level of competency, but the time it would take them to reach the required level would vary [10].

Several researchers have investigated the use of mastery learning for introductory programming [14, 17, 27, 36, 45, 49, 87]. In one study, mastery learning was used effectively with heterogeneous groups, and demonstrated an increase in student self-perception of programming ability [14].

A study of mastery learning in a CS1 course provided evidence that students who had learned using mastery learning were better prepared for a subsequent course compared with students who had experienced more traditional learning, and that the effect was particularly pronounced for weaker students [45].

Researchers investigating the use of mastery learning in a second-year software development course reported three major differences from more traditional delivery [27]: students produced code with improved design and refactoring; understanding of complex concepts such as concurrency improved; and while students did not progress as far with projects, their average grade was approximately the same.

Mastery learning has obvious applications for the design of intelligent tutoring systems for teaching programming [17]. Such systems require a model of what the student knows, and developing that model requires a detailed list of expected outcomes. Breaking up CS1 assessments into component elements may have significant implications for the modelling of student knowledge in intelligent tutoring systems designed to teach programming.

2.2.2 Keller Plans.
Keller [33] introduced what has come to be known as the "Keller Plan" for course design. Strongly influenced by his observations of training in the military, he designed a radically different version of an introductory psychology course.

The material in the course was divided into 30 units. At the start of the course, each student was given reading material and some questions to think about in relation to the first unit. When ready, he or she took a test on the material (at one of a set of possible times during the week), a teaching assistant (TA) graded it on the spot, and the student and TA had a conversation about the results. If successful, the student continued to the next unit; otherwise, he or she was allowed to be re-tested on the material an unlimited number of times. TAs, generally students who had received an A

in the course a year or two earlier, were each assigned ten students to work with, so all of a student's tests were given by the same TA. At the end of the semester, a student's grade was based on the number of units completed, plus performance on a final exam (which included questions from the individual units).

Keller's course design generated a lot of interest during the 1970s and 1980s. Interest seems to have fallen after that point, but has revived recently. It has been suggested that some of Keller's ideas might be incorporated into distance learning, and specifically, MOOCs [28].

The approach that we describe here, of breaking complex programming tasks up into individual components, may provide the basis for determining a structure for Mastery Learning or activities to be used in a Keller Plan. However, it may also inform the structure of learning in other pedagogies, such as the order in which standard lecture courses progress, or it may form the basis for multiple-choice questions in a peer instruction course.

2.3 Syntax and Semantics (RQ1)

2.3.1 Syntactic and Semantic "Elements". The elements of syntax are essentially the set of basic atomic symbols and words, as well as the rules used to assemble them into well-formed constructs. To a large extent, they are programming language dependent and, in du Boulay's terminology [23], relate to the *notation* area.

The elements that pertain to the semantic sphere can be divided between operational aspects – how the abstract *notional machine* [23] works – and task-oriented features – what is meant to be achieved by using a given program construct. Students' understanding of operational semantics is usually connected to their ability to trace a program's execution, which has been investigated in terms of *mental models* (e.g., models of assignment and recursion) [11, 37, 43, 66], and often addressed with the aid of visualisation tools [80].

The *purpose* of specific program structures, on the other hand, has been given limited attention at an elementary level. Remarkable exceptions that can be mentioned in this respect are code analysis in terms of *programming patterns* [60], and particularly of the *roles of variables* [12, 63], which seem to capture very basic aspects of the development of simple programs.

2.3.2 Learning Outcomes. To the best of our knowledge, the range and consistency of expected learning outcomes for CS1 across a variety of institutions and countries have not yet been investigated systematically. Dale [19], Schulte and Bennedsen [67] and, more recently, Davies et al. [20] provided some statistics on concepts or techniques covered as well as on programming languages in use, based on the information gathered from respondents to surveys. The trend of programming languages adopted in the US for introductory courses has also been monitored by Siegfried et al. [72], while de Raadt et al. [22] analysed 49 widespread textbooks in terms of broad topics, language, and other features.

The 2001 report of the Joint ACM & IEEE-CS Task Force on Computing Curricula suggested a set of core topics and learning objectives for the "Programming Fundamentals" (PF) unit [3] (whereas the most recent report [4] doesn't address introductory programming at the same level of detail). The core topics were organised into five areas: fundamental programming constructs (PF1), algorithms

and problem solving (PF2), fundamental data structures (PF3), recursion (PF4), and event-driven programming (PF5). PF1 and PF2, in particular, include very basic syntactic and semantic concepts which apply to most programming languages.

2.3.3 Assessment Analysis. A substantial body of work has previously explored the assessments that are used in novice programming courses [32, 58, 69, 76, 91], and the complexity of those assessments [32, 42, 90]. Assessment tasks were found to be more complex than academics expected [92]. In previous work, Luxton-Reilly and Petersen investigated the number and types of syntactic elements found in programming assessments [42]. They generated the Abstract Syntax Tree for CS1 exam questions that involved either reading or writing Python code. The elements of the syntax tree were categorised and tabulated, revealing that exam questions contained an average of 10.85 different *concept components*. They argued that current assessment practices do not target specific syntactic elements; instead, most assessments they analysed involved a large number of syntactically derived "concepts."

2.4 Relationships between Elements (RQ2)

The occurrence and strength of coupling between different concepts implied by a given task can be connected with the levels of the SOLO taxonomy [18, 39, 69, 71]. Elements that can be understood in isolation pertain to the *uni-structural* level, whereas *multi-structural* tasks involve different components that are only loosely coupled to each other. More complex tasks still can be described at the *relational* or *extended abstract* levels.

Assessments that novice programmers are expected to complete, such as those used by the McCracken working group [47] and the Leeds working group [38], are typically expressed at the relational level and require an understanding of the relationships between elements. Yet there is a growing body of evidence that students typically do not attain these levels at the end of CS1, but instead tend to be operating at or below the multi-structural level [39].

The relationships between elements may also explain why introductory programming courses have "higher than usual rates of both failing and high grades, creating a characteristic bimodal grade distribution" [61] (although the "bimodal" description is not accurate [1, 55]). The Learning Edge Momentum hypothesis proposes that success in acquiring one concept makes it easier to learn other closely linked concepts, whereas failure makes it harder [61]. This interaction between the way that students learn and the tightly integrated nature of the concepts comprising a programming language creates an inherent structural bias in CS1 that drives students towards the extreme outcomes often observed. If this theory is correct, then we may be able to promote more positive outcomes for students by replacing negative feedback on multi-concept assessments with more easily achieved positive feedback on smaller atomic concepts.

2.5 Independent Assessment (RQ3)

Computer science educators have studied the assessment of individual core concepts in recent years [85]. Questions in a concept inventory are intended to measure students' understanding of key concepts. Creation of such an inventory involves a complex process of identifying a set of misconceptions and devising corresponding

items that test for these misconceptions; in particular, the resulting items have to be empirically validated in order to ensure that they are indeed measuring the misconceptions and concept understandings that they are intended to measure. So a concept inventory is derived from a top-down approach and is based on the structure of the subject itself (for deriving concepts) and empirical investigations (for deriving/validating misconceptions). Once a concept inventory is complete, it can be used to diagnose students' misconceptions with regard to the concepts used. While there is some recent work on concept inventories in CS (see Taylor et al. [85] for an overview), the existing inventories are proprietary and unavailable [25, 26], at a pilot stage [13], or avoid syntactic issues [25, 26, 53].

The most developed instruments, the FCS1 and SCS1, are based on pseudocode. Tew developed and validated the FCS1 to be a language-independent assessment of CS1 [25, 26]. The assessment is based on an analysis of typical textbooks, and many of the concepts (e.g., assignment, loop, etc.) are the same as or similar to those that we have identified at the level of programming language elements. A second, isomorphic assessment (the SCS1) was developed to replicate the previous FCS1 and to make it available to the community [53].

Mühling, Ruf, and Hubwieser developed an instrument based on a set of Rasch-scaled items [51] that assess learners' abilities concerning control flow and control structures. It is explicitly designed to leave out aspects of program state and syntax.

Deriving a set of criteria that can be used for grading is a process that is usually done manually and often holistically. Berges and Mühling describe a way of deriving a series of questions that can be associated with a concept and then used as criteria for checking whether a concept has been understood based on source code [7]. We use a similar approach in this work to derive the atomic assessment tasks for a concept.

3 CONCEPTS FROM LITERATURE – RQ 1.1

3.1 Methodology

We began by searching the literature for a hierarchy of concepts in introductory programming courses. No papers were found that presented a concept hierarchy for CS1. Návrat [52] proposed a preliminary and partial concept hierarchy for collections, a topic that is typically covered in CS2. His focus was to distinguish between design tasks involving abstraction and generalisation using these hierarchies, and so is not directly relevant to the current project. Some papers empirically established skills hierarchies based on an analysis of assessment items [40, 74, 88, 93] while others listed programming concepts and grouped them under various headings [2, 19, 48, 65]. In a study of exams, Simon et al. [74] noted that "topics that follow 'assignment' tend to subsume data types & variables" and also in their encoding method that "Having assigned [a broader topic] to a question, we would not also assign a topic subsumed by that broader topic," giving an example of loops subsuming operators; this suggests that they believe that some of the concepts can be arranged hierarchically.

A further search of the literature, to establish the core concepts of CS1 and the key expectations for a CS1 course, identified seven papers [2, 25, 31, 56, 64, 67, 74]. In each of these papers a concept list was composed from different sources. Armstrong [2] analysed forty

years of literature in order to establish a list of essential elements or "quarks" that are key to teaching object-oriented programming. Eight quarks were identified and grouped according to whether they were structural or behavioural program elements. Schulte and Bennedsen [67] surveyed teachers in order to establish what concepts teachers considered to be most important in a CS1 course. The concepts in their survey were derived from lists collated by Dale [19] and Milne [48]. Dale's list was developed through a survey which asked teachers of introductory computer programming to identify the topics which they covered in their courses. Milne on the other hand surveyed students and built a list of concepts ranked by student perceived difficulty. Tew and Guzdial [25] surveyed four introductory programming textbooks to establish their list of common fundamental CS1 concepts. Pedroni and Meyer [56] examined the structure of dependencies of topics in OOP and, as an artefact of this study, identified a set of core OO concepts that were extracted from course design materials. Hertz [31] constructed a list from the LACS curriculum [16] by combining, generalising, splitting, and adding new concepts. Simon et al. [74] examined 20 CS1 exams and for each question in the exams identified up to three central concepts assessed by those questions. An ITiCSE 2013 Working Group [64] developed a set of 654 multiple-choice questions, the Canterbury QuestionBank, covering various CS1 and CS2 programming topics. Each question in the bank was assigned up to three topic or concept tags from a list that was based on that of Simon et al. [74] but substantially expanded to include CS2 topics and other topics that arose through the question bank tagging process.

In order to ensure a comprehensive picture of what are considered to be core introductory programming concepts, we then added the LACS concept list [16] and the Fundamental Programming Concepts and Fundamental Data Structures from the Software Development Fundamentals (SDF) area of the 2013 ACM curriculum [4].

A master list of concepts was compiled from these nine sources. Common entries were identified and either merged or split. For example, Schulte and Bennedsen's [67] "Obj&Class – The concept of objects and classes" was split into two separate categories – objects and classes – whereas "iteration" and "iterative control constructs" were merged into a single category. Some concepts were eliminated from our list: "Object-Oriented Basics" was considered to be so abstract that it was inherently covered by other lower level concepts in the list, and it also lacked clarity – which OO concepts are considered to be basic? "Design by contract" was among concepts that were eliminated because while they are important, they are not directly related to syntax and semantics and are rarely covered in an introductory programming course. Other topics such as "ethics," while interesting in a CS1 course, are not typically covered in introductory programming or related to programming syntax and semantics. Finally any topics considered to be more typically taught in a CS2 course were discarded. The remaining concepts were grouped into categories, many of which are the same as broader concepts already identified from our literature examination.

Concept	Freq	Source
Variables	6	4, 25, 31, 65, 67, 74
Scope of variables	6	2, 25, 56, 64, 67, 74
Assignment	4	4, 25, 56, 74
Expressions	4	4, 16, 31, 74
Constants	3	64, 67, 74
Instance variables	2	25, 67
Lifetime	2	64, 74
Static & non-static variables	1	67
Data Types	4	16, 31, 64, 74
Strings	6	16, 25, 31, 64, 67, 74
Primitive data types	3	4, 25, 56
Floating Point	2	25, 64
Integers	2	25, 64
Pointers, references & memory		
management	2	64, 67
Booleans	1	25
Data Structures		
Arrays	6	16, 25, 31, 56, 67, 74
Linked list	3	56, 64, 67
Collections other than arrays	2	64, 74
Matrices	1	16
Lists	1	31
Sets, relations	1	16
Structs & Records	1	64
Control Constructs	2	16, 31
Conditional Control Structures	5	4, 25, 56, 67, 74
Iterative Control Structures	4	4, 64, 67, 74
Loop (for)	1	25
Loop (while)	1	25
Loop (nested)	1	25
Events	2	64, 74
Exceptions	2	16, 65
Arithmetic	2	25, 64
Relational operators	3	25, 64, 74
Logical operators	3	25, 64, 74
Boolean algebra	1	16
Functions, Methods		
& Procedures	6	2, 4, 16, 31, 64, 74
Function Definition	1	25
Parameters & param. passing	2	25, 74
Subroutines	2	31, 74
Accessor Methods	1	25
Mutator Methods	1	25
Return values	1	25
Static & non-static methods	1	67
Calling Functions		
Recursion	7	16, 25, 31, 56, 64, 67, 74
Parameter passing	3	25, 67, 74
Dynamic binding	2	2, 56

Concept	Freq	Source
I/O	2	64, 74
Simple I/O	5	4, 25, 31, 64, 74
File I/O	4	4, 16, 64, 74
Streams	2	4, 16
GUI	2	64, 74
OO Concepts	2	64, 74
Classes	6	2, 16, 25, 31, 67, 74
Encapsulation/information hiding	4	2, 25, 67, 74
Inheritance	5	2, 16, 25, 31, 67
Polymorphism	5	2, 25, 56, 67, 74
Objects/Instances	4	2, 25, 67, 74
Interfaces (Java)	3	16, 31, 64
Constructors	2	25, 74
Generics	2	56, 67
Message passing/object interaction	2	2, 67
Abstract classes	1	31
Instance variable types	1	67
Object identity	1	74
Programming process		
Programming styles/standards[a]	4	16, 31, 64, 67
Reading code	4	16, 31, 64, 74
Debugging	2	2, 67
Design - classes[b]	2	64, 67
Design - methods[c]	2	64, 74
Design - programs	2	2, 67
Design - algorithms	1	67
Design - single class	1	67
Error handling	1	67
Testing	1	64
Abstract programming thinking		
Judgement[d]	1	16
Notional machine[e]	1	16
Problem solving strategies[f]	3	31, 64, 74
Using language libraries	2	65, 67

[a]Complying with a set of programming style guidelines or standards
[b]Identifying classes from a problem description. Schulte and Bennedsen [67] also include CRC cards and responsibility design in their list as a separate category.
[c]Given the classes needed to solve a problem specify the methods required
[d]Choosing appropriate data structure, algorithms and being able to justify that choice.
[e]Mental model of the computer.
[f]Including decomposition of a problem (divide and conquer), stepwise refinement and other problem solving strategies.

Table 1: Frequency of common CS1 concepts

3.2 Results and Discussion

Table 1 presents the final master list of concepts extracted from the nine literature sources. It is worth noting that the disparate approaches used to develop the nine concept lists influences the types of concepts identified in each list. For example, those developed from a survey of teachers [67] focus more on classes of syntactic concepts (e.g., variables and data types, data structures, etc.), processes (e.g., design and testing), whereas those developed from examining assessment items tend to focus on individual concepts. When concept lists focus on syntax and semantics, the problem of language specificity is introduced.

In undertaking this analysis of concepts it is clear that concepts are unlikely to fall into a strict hierarchy – it is more likely that there is a myriad of connections (a graph) with horizontal and vertical relationships. For example, the concept of "scope" arises in several lists. This might refer to any of several higher-level concepts, such as variable scope, method or function scope, or scope of any of those constructs as specific to OO programming.

4 LEARNING OUTCOMES – RQ 1.2

4.1 Methodology

We analysed the CS1 learning outcomes of 103 courses[1] from 101 universities in 12 countries: Australia, Austria, Canada, France, Germany, Ireland, Italy, New Zealand, Spain, Switzerland, the United Kingdom, and the United States. To select these courses we first used the Times Higher Education ranking (THE)[2] to generate a list of the institutions in each country that offers CS bachelor's degrees. For each country we then randomly selected ten institutions that both offer an introductory programming course and publish the learning outcomes for that course on the web. If a country had fewer than ten suitable institutions, we included all that it had. We found two universities that offered more than one introductory programming course; in those cases, we included both. A detailed breakdown of the countries and courses is shown in Table 2.

Two of the researchers analysed all of the learning objectives for the 103 courses in our dataset. We performed a deductive content analysis [44], using a set of twelve broader concepts based on the list of concepts in Section 3: *Variables* (including constants, expressions, assignment, and scope); *Data Types*; *Data Structures*; *Control Structures*; *Functions/Methods/Procedures*; *Input/Output*; *Object-Oriented Concepts* (including classes, encapsulation/information hiding, inheritance, polymorphism, objects/instances, Java interfaces, constructors, message passing/object interaction, etc.); *Recursion*; *Memory Management/Pointers/References*; *Libraries*; *Programming Process* (specifications, design, coding, testing, debugging, documentation), and *More Abstract Thinking about Programming* (for example, correctness, algorithm analysis, judgement, and problem-solving strategies).

For training, we independently tagged a sample of 10% of the data. We agreed on 96% of our ratings, with a Cohen's Kappa of 0.813. After discussing and resolving our differences, we agreed on a set of guidelines and each rated the full dataset.

Country	Institutions	Courses
Australia	10	10
Austria	3	3
Canada	9	9
France	8	8
Germany	10	10
Ireland	8	9
Italy	10	10
New Zealand	7	8
Spain	10	10
Switzerland	6	6
United Kingdom	10	10
United States	10	10
TOTAL	101	103

Table 2: Total courses in learning objective sample, by institution and country

Relative to the whole dataset, we agreed on about 96% (again) of our ratings, with a Cohen's Kappa of 0.772. On individual concepts, the percentage of agreement varied from about 78% (Programming Process) to almost 100% (Input/Output, Recursion, Memory Management).

There were 540 learning objectives in total, or an average of approximately 5.2 per course. The number of objectives varied widely, however, from a minimum of one to a maximum of 22 for a single course. In addition, the division of course objectives into individual objectives seemed somewhat arbitrary.

Accordingly, we computed an aggregate result for each course, based on the results for all of its individual learning objectives. Specifically, considering the tags for each learning objective as a bit vector (with 1's for the tags that apply to that learning objective and 0's otherwise), the course result is the "or" of the bit vectors for all of its learning objectives. For example, if any of a course's learning objectives was tagged as having something to do with recursion, the course as a whole was assigned a 1 for recursion.

Finally, we reconciled the differences in the individual tags as far as necessary to reach agreement on the course-level results. When we reached a point where there were only 86 differences left to reconcile (about 1.6% of all the tags), none of the remaining tags would make any difference at the course level. For example, if a course has three learning objectives, one of which clearly refers to recursion, that course includes recursion in its overall objectives regardless of the other two objectives. We stopped at that point.

Besides looking at concepts, we also noted any learning objective that mentioned a specific programming language. This was very straightforward.

4.2 Results and Discussion

Based on our stated learning objectives, what *do* we expect from our CS1 students? The concepts addressed in the learning objectives of our sample courses are shown in Table 3.

---

[1]We use the term "course" to refer to a unit of instruction, typically a semester in duration, for which a student can earn a final result, and which is known variously as a course, a module, a unit, a subject, a paper, and perhaps more.
[2]https://www.timeshighereducation.com/world-university-rankings

Concept	Number (%) of courses
Variables, etc.	14 (14%)
Data Types	24 (23%)
Data Structures	41 (40%)
Control Structures	34 (33%)
Functions, Methods, Procedures	27 (26%)
Input/Output	18 (17%)
Object-Oriented Concepts	37 (36%)
Recursion	10 (10%)
Memory Management, Pointers, References	5 (5%)
Libraries	15 (15%)
Programming Process	90 (87%)
More Abstract Thinking about Programming	65 (63%)

Table 3: Number of courses whose objectives addressed each concept (with percentage in parentheses)

The most common concept, Programming Process, is found in 87% of courses, substantially more than the next highest (More Abstract Thinking about Programming, 63%). The prevalence of these two concepts indicates that, as a community, we overwhelmingly expect students who have completed CS1 to know how to program and to be able to reason about programs. By contrast, the concept "Memory, Memory Management, Pointers, and References" was referred to in just 5% of the courses.

The stated objectives for CS1 are often fairly abstract. In some cases, we concluded that they were too vague and generic to tag; for example CA_09's only learning objective reads:

> Write programs to solve problems in computer science

Learning objectives that address only Programming Process are clearer, but still quite abstract. Consider the only objective for course AT_01:

> Understand systematic, constructive approaches for creating, testing, debugging, understanding and modifying programs

Clearly, this course is focused on Programming Process. But consider what the objective does *not* do: it does not explicitly address any of the more specific concepts in our list, such as control structures, functions, or libraries. From a pragmatic point of view, this makes sense: the objective might remain valid for decades. But it is challenging to make a connection between such high-level objectives and the atomic assessments that are the focus of this paper.

Objectives related to More Abstract Thinking about Programming are also fairly abstract; see, for example, Learning Objective AU_06.04 (the fourth objective of course AU_06):

> Discuss the rationale for applying particular programming concepts

On the other hand, we did find more concrete learning objectives, such as AU_04.06:

> Write Java code that uses control structures, classes and arrays.

CA_06.13:

> Develop and use boolean expressions comprising relational and logical operators.

and CH_01.02:

> Know a part of Java's class library.

For an example of how a question related to the second of these learning objectives (CA_06.13) might be broken down into atomic elements, see Section 7.2.

We have defined as "highly abstract" those courses whose aggregate learning objectives

- *only* address the Programming Process and/or More Abstract Thinking about Programming concepts, or
- are too vague even to categorise.

Overall, 27% of the courses fall into this highly abstract category. These courses will be more difficult to connect with atomic assessment items.

Finally, in our examination of learning objectives, we found references to two additional concepts, one of which is not covered at all in Section 3's list of concepts, and one of which is briefly implied. The first of these is *translation (or mapping)*. Translating from one representation to another equivalent one is a key skill for computer scientists: for example, translating from UML to skeleton code, from logic expressions to circuit diagrams to truth tables, from box-and-arrow diagrams to linked-list code, from tail recursion to iteration, from a recursive function to a program that applies memoisation, or from binary to hexadecimal to decimal number representations. Although this concept was not found in the literature, it is concrete enough to be assessed. If it is not atomic, it is considerably closer to atomic than an objective such as "write programs to solve problems in computer science" (CA_09, discussed above).

Several courses included examples of translation in their learning objectives. For example, CA_08.04 features flow charts:

> Translate a simple pseudocode or C program into a flowchart.

UK_04.04 focuses on UML:

> Interpret UML class diagrams in order to implement object-oriented software

Additional examples of translation or mapping found in our learning objectives include modelling real-world situations in Java, translating algorithms into code, translating functional requirements into code, and being able to express algorithms in pseudocode and/or flowcharts. Besides being able to do such translations themselves, students are also expected to understand that the compiler translates their programs into an executable form (UK_09.03).

The second concept that should perhaps be added to our list is *composition*: the ability to combine atomic constructs. This is implicit in Section 3's mention of "nested loops", but it occurs in many other contexts as well. As a matter of fact, instances of this concept are implied by any interesting combination of basic control structures – as well as the aggregation of data into structured data types – and it is connected with the approach to task decomposition discussed in Section 7. Nevertheless, the ability to understand the effect of such combinations, also in connection with possible syntactic nuances, is seldom recognised as a key topic in itself, although students often struggle with nesting conditionals and loops ([35]). Moreover, if the approach that is central to this paper – devising

Programming language	Courses	Percentage
Java	38	36.9%
Python	21	20.4%
C	17	16.5%
C++	10	9.7%
Scheme/Racket	4	3.9%
C#	2	1.9%
Caml	2	1.9%
Javascript	1	1.0%
Pascal	1	1.0%
Processing	1	1.0%
Unknown	13	12.6%

Table 4: Programming languages in the sampled universities; percentages sum to more than 100% because seven courses specify two programming languages

atomic assessments for at least some programming concepts – is to succeed, it is essential that students then be able to combine the concepts that they have learned.

In addition to looking at concepts mentioned in the learning objectives, we also retrieved information on the programming languages adopted in the considered courses. This information was publicly available for 90 of the 103 courses (87%). Table 4 summarizes the use of different programming languages in the sampled universities. Percentages sum up to more than 100% because in 7 courses a second programming language is introduced. It may also be worth noting that 43 courses (42%) make explicit reference to some specific programming language in the learning outcomes themselves, and one more course mentions C as a possible (suggested?) option. As mentioned above, the concept "Memory, Memory Management, Pointers, and References" is referred to in only 5% of the learning objectives; given the frequency of C and C++ in this picture, this result is even more surprising.

5   ANALYSIS OF ASSESSMENTS – RQ 1.3

In the previous section we explored what topics students are exposed to in CS1 courses. In this section we ask whether the assessments set by instructors actually assess the topics in the learning outcomes, and whether they successfully assess these outcomes in isolation.

5.1   Methodology

Luxton-Reilly and Petersen previously analysed the content of CS1 exams in the Python language, with the aim of identifying the elements of syntax that were assessed and the extent to which those elements were assessed in isolation [42]. To determine whether their results hold in a different context, and to gather more evidence of the range of concepts that computing educators evaluate, we replicated their analysis using a different data set and incorporating a second programming language.

Our data set was drawn from the Canterbury QuestionBank[3], a repository of multiple-choice questions developed by a 2013 ITiCSE

---

[3]http://web-cat.org/questionbank/

working group [64]. The QuestionBank was intended to contain questions for both CS1 and CS2, which is a threat to validity: the questions do, in aggregate, cover more material than is covered in a typical CS1 course. As a result, we would expect the number of concepts in an average question to be higher than in the original work.

The Canterbury QuestionBank contains 656 multiple-choice questions, and we extracted all of the code from every question that included syntactically valid code, whether it was an entire program or a segment. This entailed substantial data cleaning. The questions are available in multiple forms, including as web pages (html) and as an xml file of some 115,000 lines. We understand that these were both scraped from PeerWise[4], in which the questions were originally submitted. As a general rule, code segments within questions and answers are enclosed between *pre* and */pre* tags, which mark out the code and indicate the language in which is is written. Unfortunately, the scraping did not apply this rule consistently. Some code segments are not marked at all, while others have some lines of code marked and others not. Whether marked or not, some code also has interpolations that render it syntactically incorrect, such as explicit line numbers or indications of the location of missing code.

On initial analysis it appeared that only 236 of the 656 questions included program code. Exactly half of these questions, 118, required editing to ensure that all of the code was tagged consistently and could be parsed. A further 181 questions were found that included code that had not been tagged at all. Once these were edited and tagged appropriately, we had 417 questions incorporating tagged code that could be parsed and analysed. The questions together encompassed five programming languages: Java (301 questions); C (72), Python (36), Perl (5), and Visual Basic (3).

We confined our analysis to the Java and Python questions, using a separate script for each. For each question in turn our scripts analysed all of the code, which might be incorporated in the question, the answers, or both. The scripts generated the abstract syntax tree for each piece of code and then traversed the tree, forming a set of the syntax elements encountered. Some features of the code, such as whether a variable was being declared or used, were inferred and included in the set. After all of the questions were analysed, the total numbers of occurrences of each syntax element (or inferred feature) were tabulated.

5.2   Results and Discussion

At a high level, our results echo the original analysis [42]. The mean, minimum, and maximum number of concepts in each set are presented in Table 5. While there were few Python problems in the Canterbury set, the average number of concepts in the two Python datasets was similar (10.91 vs. 10.85). The Java questions in the Canterbury set involved more concepts on average (12.83).

The slightly higher number of concepts in our set of Java problems may be a result of the language itself. To gauge how large an impact language is on the number of concepts found, we removed syntax related to classes (the keyword *class* and annotations for visibility like *public*) and related to types. The average number of concepts per question dropped to 9.83, indicating that the language

---

[4]https://peerwise.cs.auckland.ac.nz/

used has a clear effect. However, the difference in average number of concepts is possibly also influenced by other factors, such as the inclusion of CS2 questions in the set or a difference in the style used to ask questions. As an example of the latter, in the original Python dataset, *print* was frequently used (in 65% of problems), as students were asked for the output of a piece of code. However, it was used less frequently in both our Java (35%) and Python (43%) sets. As a result, the relative appearance of concept items is likely to be a more effective means of comparison than the average concept count.

Dataset	Number of problems	Concept Components		
		Mean	Min	Max
Original (Python)	195	10.85	2	23
Canterbury (Python)	36	10.91	2	23
Canterbury (Java)	301	12.83	1	26

Table 5: Overview of concept component counts

Original (Python)		Canterbury (Python)		Canterbury (Java)	
component	%	component	%	component	%
Variable	99.6	Variable	94.3	Variable	94.0
=	85.8	Func Call	77.1	=	75.1
Func Call	84.6	Numbers	77.1	Func Call	70.4
Numbers	83.8	=	68.5	Numbers	70.1
Sequences	81.7	Sequences	62.9	.	70.1
print()	64.6	if	57.1	Var Decl	69.4
Func Decl	54.2	+	48.6	int	62.1
+	45.4	else (if)	45.7	public	53.8
for	38.3	print()	42.9	Func Decl	46.8
if	34.6	<	37.1	if	40.5
return	34.2	Func Decl	34.3	<	36.8
.	30.4	[]	34.3	++	33.5
[]	30.0	len()	28.6	print*	34.5
len()	20.8	for	28.6	return	33.2
else (if)	20.0	while	25.7	String Lit	31.9
range()	19.2	return	25.7	+	30.5
<	16.7	input()	22.9	class	28.6
+=	16.7	True	20.0	for	28.2
>	14.2	==	20.0	[]	27.9
==	14.2	- (subtraction)	20.0	new	26.2

Table 6: Top 20 concept components from the original syntax element study [42] and a replication using the Canterbury Question Bank

The first two columns of Table 6 compare the most frequently observed concept items in the Luxton-Reilly and Petersen's dataset to those in our Python dataset. Sixteen of the top 20 "concept components" in their analysis are also in the top 20 in our dataset, suggesting that syntax-based analysis of concepts may be stable across datasets. However, the order of elements does differ, reflecting differences in emphasis in the two sets. In the Canterbury

questions, for example, *if, else,* and *<* are more common, suggesting that relatively more questions focusing on conditionals were found in the Canterbury set, while *def* and *return* are less common, indicating a lower emphasis on functions.

The third column of Table 6 lists the most frequently observed concept items in the Java dataset. Thirteen of the top 20 "concept components" in the original analysis of Python problems were found in the top 20 in our analysis of Java problems, indicating substantial overlap in the structures used in CS1 problems across these two languages. The items that were not found are missing because of differences in the language. In Java, we saw elements related to typed and declared variables ("Variable Declaration" and *int*), which do not occur in Python, and to classes (*public* and *class*), which are frequently taught later in Python. We also saw *++* in Java, while in Python we saw a structure used in a similar manner: *+=*. The items in the Python list that were not found in the Java list are either language specific ("sequences," which are a builtin type in Python, and the *range* function) or were displaced from the top 20 by class and type elements, but are still found in the top 30 Java concepts.

This reanalysis raises three interesting points. First, the number of syntactically derived concepts in a typical question asked of CS1 students is as high as initially reported [42], providing some validation of previous findings. In Section 7 we will show that the AST-based analysis identifies elements similar to those that were manually identified by experts who looked at the individual concepts required to answer a given problem; as a consequence, this approach suggests that automated analysis may be an effective approach for identifying a minimal set of concepts required to solve programming problems.

Second, the syntactic composition of typical CS1 problems is relatively stable regardless of language or context. We found substantial overlap in the components identified in their analysis as well as their relative ordering. The format or style of questions asked does influence the content of questions, as we can see by looking at the relative rate of appearance of *print* operations and function-related structures in our three data sets. The programming language of instruction also has an effect, as seen by the appearance of class-related items in the Java set. However, fundamental procedural structures are very common across sets.

Third, the high rate of appearance of some syntactic structures (such as variables and function calls) suggests that there are dependencies between syntactic elements that may not permit the independent evaluation of some elements. To further explore this question, we analyse dependencies suggested by the language itself in Section 6 and then manually identify relationships between concepts in example questions in Section 7.

6 WHAT ARE THE RELATIONSHIPS BETWEEN THE ELEMENTS? – RQ 2

In the preceding section we saw evidence that instructors do not assess syntactic elements independently. It is clear that the languages we use require certain elements to be used together, which probably encourages instructors to combine these elements in assessments. In this section we perform an analysis of a Java grammar to explore how the restrictions imposed by the language may influence

the questions that we use to assess students. In this section we describe an automated effort to identify relationships between syntactic elements; in Section 7 we involve experts to discover similar relationships; and in Section 8 we compare the results.

6.1 Methodology

We derived a graph of language elements from a version of Java's grammar [54]. To produce the graph, the grammar was parsed, and the nodes below the idea of a "block" were extracted. This generates a graph that contains the ideas of "statements" and "expressions", the procedural elements of the language. Class declarations and package imports are omitted. We expect that performing a similar analysis on Python's grammar would generate a similar graph.

After extracting the subgraph for statements and expressions, we made a number of changes to the grammar to improve clarity: a few nodes were renamed ("genericInvocation" to "functionCall"), nodes that were introduced to remove grammar ambiguities were merged (since we don't actually need to use the grammar to parse Java), and nodes or edges representing items not typically covered in CS1 were removed (advanced annotations and `synchronise` blocks).

6.2 Results

Figure 1 illustrates the relationships between different grammatical elements. The structure of the figure suggests a hierarchy of topics that need to be covered. However, the hierarchy does not require that all topics lower in the hierarchy be mastered before advancing to a given node. Instead, in many cases, only a single path up the graph is required. For example, to be fully exposed to the idea of a literal, a student would need to learn about six types, but to use a literal in an expression, the student need only have experience with one of those types. A single path is not always sufficient: for example, to use a method call with arguments a student will need to have been exposed to both identifiers (the method name) and arguments.

Category	Syntactic elements
Arithmetic	(...), ++, −, +, -, *, /, %
Arrays	[...]
Assignment	=, +=, -=, *=, /=, %=, &=, ˆ=, >>=, >>>=, <<=
Bitwise	&, |, ˆ
Boolean	~, !
Casting	(*type*)
Relational	==, !=, <=, >=, >, <
Logical	&&, ||
Objects	., instanceof, new, this, super
Shift	<<, >>
Ternary	? :

Table 7: Syntactic elements included in an expression

Two nodes, "statement" and "expression", encapsulate a large number of syntactic elements. The former was expanded because different statements, such as try-catch blocks and return statements, have different structures. All of the keywords associated with statements (such as `for`, `try`, and `break`) were found in the Canterbury

question bank except for `assert`. We chose not to expand the expression node; instead, the syntactic elements encountered in that node are categorised in Table 7.

Most of the items in the table are likely to be encountered in a CS1 course and were, in fact, found in the questions extracted from the Canterbury question bank. The items that were not found are the bitwise, shift, and ternary operator categories. The `instanceof` operator was also not seen, although the other members of the Class category were. Several of the augmented assignment operators were also found in the question bank, although they were used in a different context: assignment *expressions* are inherited from C and are unlikely to be taught in most CS1 courses.

The dependencies enforced by the grammar of the language constrain how we introduce new material, particularly if we want students to master a concept before reading or writing code that introduces another *related* concept. Pathways through the graph may suggest opportunities to introduce material, and develop scaffolded tasks, while minimising extraneous load.

7 HOW MIGHT WE ASSESS THE ELEMENTS INDEPENDENTLY? – RQ 3

7.1 Rationale and Methodology

Here we present a contrasting approach to the top-down derivations of a concept inventory described in Sections 3 and 4. In this approach our result is derived in a bottom-up way from the programming language as a concrete entity that is the object of interest in teaching programming. For a concept inventory, the "atomic" elements that one is dealing with are the concepts, and the items for each concept are the constituent parts of these "atomic" elements (maintaining the analogy, the sub-atomic particles). For an approach that is based on the programming language itself, the atomic parts are tokens (syntactic units) of the language in isolation or combination.

Just as a concept inventory can help to pinpoint specific misconceptions that students struggle with, our approach should help in identifying which parts of the elementary syntactic and semantic elements of a programming language a student has problems with. Some of these elements, such as "variable", may correspond to a typical concept of an inventory, while others, such as "identifier", are more specific but nevertheless integral to learning how to program.

So while the atomic elements are defined rather strictly based on the grammar of programming languages, it is not quite so straightforward to identify the decomposition of these elements into "subatomic particles". It is this decomposition that we address in the following case studies. We consider an element to be atomic if it is something that can be assessed in isolation, at least to a reasonable degree. To arrive at these elements of assessment in a way that is reproducible, we looked at the syntactic elements of a program as well as the concepts derived from the grammar of the language. The dependencies derived from the grammar (see Figure 1 for Java) show that some concepts, such as "identifier", are sinks in the graph. It therefore seems reasonable to assume that those will form the very basic elements that can be assessed in isolation, whereas the more complex categories, such as "block", may be inherently dependent on the presence of other concepts – in this case "statement" –
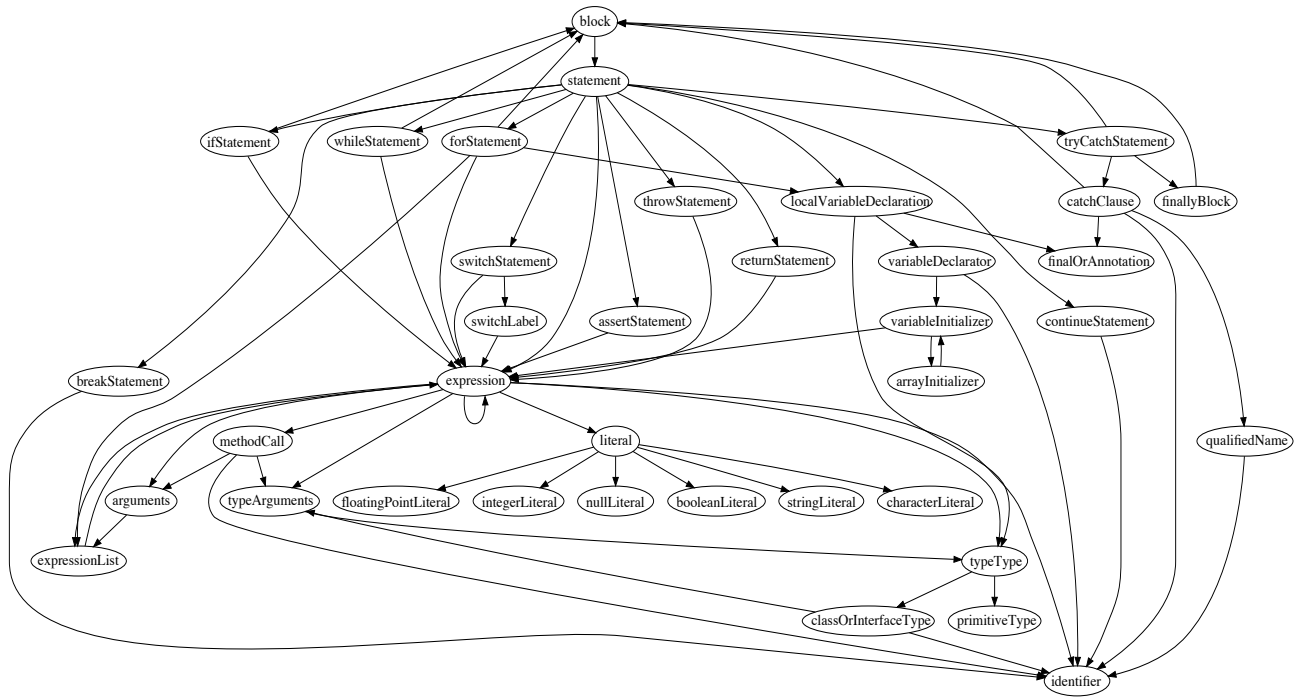
Figure 1: Dependencies enforced by the Java language

as it is hard to imagine how an assessment for a compound statement can be completely isolated from the concept of a statement. In such cases we aim for a clear indication of the constituent parts and their interdependencies to guide the process of finding a series of assessments for the more complex concepts.

By definition, deriving atomic elements based on a grammar and decomposition makes the elements dependent on the language used. The case studies will illustrate some elements that appear in Java but not in Python. Changing the language also has an influence on the constituent parts of an element. For example, some understanding of types is necessary for a variable declaration in Java, but not for Python, in which variables are not explicitly declared. The case studies will also illustrate that the constituent parts of each atomic element are dependent on the instructional approach and on the context of their appearance in the code. For example, if the instruction introduces arrays as something akin to a primitive type, not touching the aspect of references, the constituent parts of an array declaration should not include these aspects. Also, if a question is posed in a way that allows a student to answer it without having to know some particular detail, then this detail should not be included in the questions that a concept is decomposed into. However, a different question on the same concept might require this detail to be included. To continue with the example of a reference, if a question involves call by reference or copying of references, the simple notion of the variable holding the object instead of the reference will no longer work, and the idea of references must be included in a decomposition of the task.

Benchmark	Python concept count	Java concept count
Q1	6	6
Q2	4	6
Q3	2	4
Q4	7	9
Q5	5	5
Q6	13	14
Q7	12	12
Q8	8	9
Q9	11	11
Q10	14	18
Average	8.2	9.4

Table 8: Concept component counts for the benchmarking questions

This section of the work is based upon the ten benchmarking questions proposed by Simon et al. [76]. We ran the same analysis as in Section 5 on the Python and Java versions of these questions, and the resulting counts of concepts are listed in Table 8. Several of the benchmarking questions have fairly high concept counts, showing that even simple code segments can contain high numbers of syntactic items.

| | AST-based | | |
Benchmark	Python Concepts	Java Concepts	Expert-based Java Concepts
Q1	variable, numbers, and, or, <, <=	variable, numbers, and, or, <, <=	Identifier, Boolean Operator, Relational Operator, Precedence
Q4	variable, if, else, numbers, =, >, <	variable, var decl, numbers, int, if, else, =, >, <	Boolean Operator, Identifier, Assignment, Initialization, Conditional, Type, Variable Declaration, Sequence, Block Statement, Nesting

Table 9: Concept components for selected benchmarking questions

Three of the authors then acted as a team of experts to analyse two of the benchmarking questions, one that is rather simple and one that contains more complex code. A list of tokens was extracted from the code that was given in the question, the constituent parts of each token were identified, and questions were proposed that would assess a student's knowledge of each constituent part. The aim of this work is not to present an exhaustive, unique, or best set of assessment tasks, but to show as a proof of concept that code can be decomposed along the line of tokens into small – and for the most part independent – assessable components.

Table 9 summarises the concept elements that were extracted from the two sample questions both by the automatic analysis and by the expert analysis, which is described in detail in the following subsections.

7.2 Case Study 1

As a first case study we analyse in detail the first of the benchmarking questions [76] using Java as the default language. At the outset it seems to be a very simple question:

> If a dependent child is a person under 18 years of age who does not earn $10,000 or more a year, which expression would define a dependent child?
>
> ```
> A. age < 18 && salary < 10000
> B. age < 18 || salary < 10000
> C. age <= 18 && salary <= 10000
> D. age <= 18 || salary <= 10000
> ```

From a conceptual point of view it is clear that concepts such as boolean operators are relevant to this question. When analysing the tokens that appear in the code segment, we find age, <, 18, &&, salary, 10000, <=, and ||. From a semantic perspective, these correspond to the categories of identifier, numeric literal, relational operator, and boolean operator.

The goal is now to define a set of (perhaps trivial) tasks for each of those atomic categories that can be used to assess whether the student understands the syntax and semantics of that category. As described before, these tasks will vary, depending on the programming language used and on the context of token. In this specific case, the following set of tasks were identified:

Identifier In the context of this question it is not necessary to understand the complete concept of a variable; it is enough to understand that an identifier acts as a placeholder for a value in an expression and that an identifier is distinctly different from a literal. This leads to these tasks:

I1/L1 Which of the following code elements are identifiers and which are literals?
gender, 3.0, "giraffe", 5, x

I2 Can a variable hold/represent/refer to different values at different times during the execution of a program?
Yes or No

Boolean operator For the boolean operator, the students need to understand how the operator itself works and, as a prerequisite, that there are two boolean values:

BO1 List all of the boolean values

BO2 Give the results of the following expressions:
true && true, true && false, false && true, false && false

Relational operator To understand a relational operator a student must understand both its syntax and its semantics. It may be plausible to assume that this is not a problem for the operators '<' and '>' as they have the same denotation as in mathematics. However looking at the less or equal that appears in this particular question it becomes obvious that understanding the syntax of this operator is a prerequisite for understanding the code segment. It is also important to understand that comparisons work only for things that are actually comparable and that the result of a comparison is a boolean value. This leads to the following task:

RO1 Give the results of the following expressions:
3 < 4, 5 < 7, 3 <= 3

Precedence An issue that follows not directly from a specific token, but from the fact that multiple operators can appear without parentheses, is that students must be aware of the precedence of operators for the evaluation of the result. This can be tested by the following task:

OP1 In the expression *a < 10 && s > 3*, which of the operators (<, >, &&) is evaluated first *(or second, or third)*?

There are certain dependencies in these tasks. For example, to test RO1, BO1 has to be used first to establish that the boolean values themselves are known by the learners. The dependencies result both from inherent dependencies among the concepts (e.g., between identifier and variable) and from the specific assessments used. Figure 2 illustrates these dependencies.

While some items, such as BO1, are general and can be used in this particular form in every relevant scenario, others, such as BO2, can be understood as template questions. There would also be a BO3, not listed above, which uses the same approach but for the operator '||' instead of '&&'. The template question would therefore
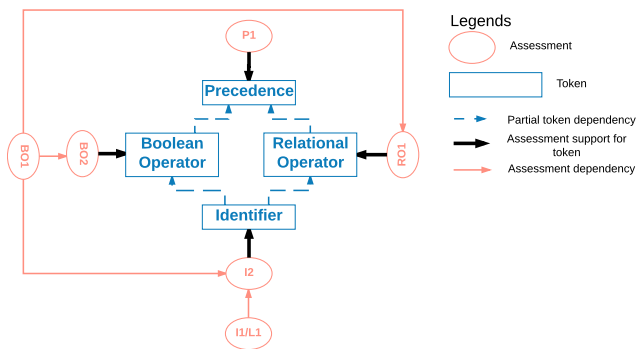
Figure 2: Hierarchical relationship between tokens and assessment questions for case study 1

be: "Give the result of the following expressions: …" This template must then be "instantiated" for every specific operator that appears in a question.

These specific tasks should serve only as examples; there may be others ways of assessing the elementary aspects and it may not even be fully established that our proposed tasks are indeed valid assessments beyond the face validity that we established by agreement among our team of experts. However, it quickly becomes obvious that even a very basic code segment can involve a complex list of assessments together with dependencies among the tasks. In this case we have arrived at seven distinct things to test, and this omits aspects that are not apparent as tokens in the code segments but are clearly a cognitive process of solving this task, such as understanding the natural language description and the relationship between "earn" and "salary".

7.3 Case Study 2

The second case study involves rather more complex code than the previous question, and is the fourth of the benchmarking questions [76]:

> What will be the value of the variable *z* after
> the following code is executed?

```
int x = 1; int y = 2; int z = 3;
if (x < y) {
    if (y > 4) {
        z = 5;
    } else {
        z = 6;
    }
}
```

The number and complexity of the syntax token combinations increases substantially with the number of lines of code. The distinct token categories that appear are: identifier, assignment, block, variable declaration, variable initialisation, and conditional. The first line alone touches on the concepts of type, variable, assignment, statement and sequence. Apart from identifier, which was discussed in the preceding case study, we can identify the following tokens:

Type As Java is a strictly-typed language, variable declarations necessarily involve types. Students must understand that values in Java have types, and that "int" denotes the specific type representing the syntactic token of an integer number. This leads to these questions:

T1 Which of the following literals are of type int?
3.2, 7, true, "a", "5"

Variable declaration The variable declaration results in a unique identifier, of a given type, to which values can be assigned. In other words, the value denoted by the identifier is mutable. This is a distinct idea from the actual assignment of a specific value to this identifier, which is covered below:

VD1 Can the value of a variable change during the execution of a piece of code?
Can the value of a variable stay the same during the execution of a piece of code?

VD2 Is the following code valid?
```
int x;
int x;
```

Assignment The assignment of a value to a variable includes the ideas that the operator is not commutative, that the types of both sides have to match, and that for primitive types the assignment copies values (i.e., the right side will be evaluated and the value will be stored in the left side).

A1 What type must the variable *x* have for the code *x = 1;* to work successfully?

A2 What is the value of the variable *x* after the code segment *x = 1;* has been run?

A3 Are the two statements *x = 1;* and *1 = x;* equivalent?

A4 What is the value of *a* after executing the following code segments, where *a* and *b* are variables of type *int*?
```
a = 1;
b = a + 2;
b = 3;
```

Initialisation Finally, the initialisation is a compound construct of declaring a variable and assigning an initial value and can be understood as such.

I1 Is the code segment *int x = 1;* equivalent to:
```
int x;
x = 1;
```

Sequence A sequence of operations includes the notion of single statements being executed in a strict sequence. Also, syntactically, it must be understood in this context that the end of a statement is denoted not by the end of a line but by a semi-colon.

S1 Is the code *int x = 1; int y = 2;* equivalent to:
```
int x;
x = 1;
int y;
y = 2;
```

S2 Will the single statements of the following code always be executed in the order that the statements are written?
```
int x = 1;
```

```
int y = 3;
x = y + 4;
```

Here we have intrinsically related concepts. Understanding that an assignment copies a value requires an understanding of a sequence of operations: evaluating the expression on the right and then copying its value to the variable on the left. Yet in our suggested questions the sequence of operations is assessed using assignments. It is possible to assess a sequence using other operations (e.g., print statements), but that would require the understanding of those other operations.

Continuing with this question, relational operators and boolean values were addressed in the preceding case study, but understanding of several other elements is required: conditional control flow, the syntax of a block in Java, and the meaning of nested structures.

Conditional As a conditional may or may not have an else, both aspects should appear in the assessment.

C1 What is the value of x after the execution of the following program segment?

```
int x = 1;
if (x > 2)
 x = 2;
```

C2 What is the value of x after the execution of the following program segment?

```
int x = 1;
if (x > 2)
 x = 2;
else
 x = 0;
```

Block statements What is the value of x after the execution of each of the following program segments?

B1
```
int x = 1;
if (x > 2) {
   x = 2;
   x = x + 5;
}
```

B2
```
int x = 1;
if (x > 2)
   x = 2;
x = x + 5;
```

Nesting The nesting of control structures is, again, a template: it can be used, for example, with loops as well as with conditionals. In this case, we stick to the specific nesting of two conditionals.

N1 What is the value of x after the execution of the following program segment?

```
int x = 1;
if (x < 3) {
   if (x < 1) {
      x = 2;
   }
   x = x + 5;
```

```
}
```

In the case of N1, we end up with a task that is almost as complex as the original question, since it comprises a nesting of two conditionals. Still, the ordering of the assessment items would give the information that if all items except for N1 are answered correctly, the student may be failing to understand that control structures can be nested, or may be failing to perceive how the nesting is expressed syntactically. Figure 3 illustrates the complexity in the relationships inherent in this question.
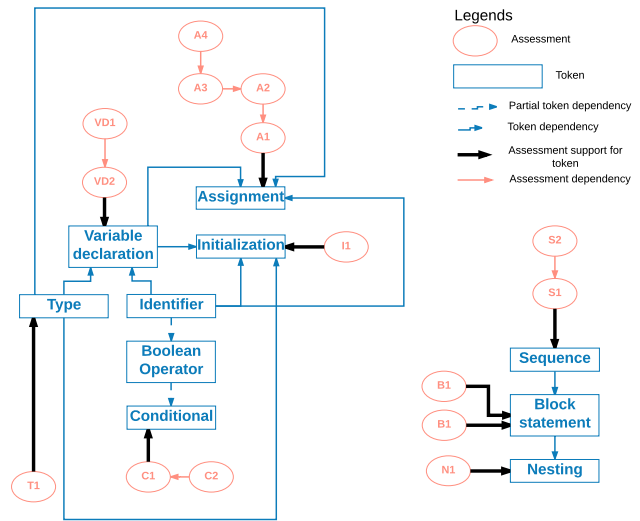

Figure 3: Hierarchical relationship between tokens and assessment questions for case study 2

7.4 Case Study Summary

The analysis above shows that a seemingly straightforward question can involve a relatively large number of tokens, and that misunderstanding any of these tokens can prevent students from answering the original question. However, the reverse is not necessarily true: a student who understands all of the tokens might not be ready to answer the original question. This is because factors beyond the scope of this very strict decomposition process may come into play, factors such as excessive cognitive load or issues with translating a problem from natural language to code.

Notwithstanding these limitations, the information that is gained from assessing the constituent parts in isolation can be valuable in cases where a student fails the original question and also fails some of the smaller component assessments. This would give a clear indication of where the student's misunderstanding might lie, thus providing feedback to both the lecturer and the student. Appendix A lists all of the concepts indicated in Figure 1, together with indicative assessment tasks.

The decomposition process outlined in the case studies can be used by educators to provide more targeted feedback to their students. They might use the Abstract Syntax Tree (AST) as a starting

point to examine possible atomic units, but other concepts not included in the AST might be needed in their final list.

It is clear to us that there is no single correct decomposition for a given question; indeed, the members of our expert group did not always agree on how a certain concept should be decomposed. For example, in question 6 from the benchmarking paper [75], the statement

```
int [] nums1 = { 1, -5, 2, 0, 4, 2, -3 };
```

led to disagreement on whether the concept of references is necessary as a token. A discussion revealed the idea that the granularity of decomposition is context-dependent, and that the concept of references is not needed in the decomposition of this particular code in the context of this particular question. This makes it clear that the decomposition process is not necessarily unique, and is likely to be subjective.

8 DISCUSSION

Our analysis of the literature in Section 3 indicated a number of common concepts, such as variables; scope of variables; assignment; expressions; data types; strings; arrays; conditional control structures; iterative control structures; functions, methods, and procedures; recursion; simple input/output; file input/output; classes; encapsulation/information hiding; objects/instances; programming style/standards; and reading code.

Although we, the authors, appear to share a common intuition about the important concepts covered in introductory programming courses, we observed highly varied learning outcomes for such courses in our sample. The learning outcomes are likely to depend, at least in part, on institutional reporting requirements and local cultural expectations. Nevertheless, the lack of consistency in learning outcomes highlights the lack of shared understanding about the appropriate *expectations of student performance* in CS1, the lack of shared understanding about the best level of detail/abstraction in describing these expectations, and the high variability in the emphasis placed on different expectations within introductory programming courses.

We note a reasonably high degree of consistency between the relationships that we discovered mechanically (Section 6) and those resulting from the analysis by experts (Section 7). This suggests that the automated approach may be plausibly used as a surrogate for manual analysis by experts, and confirms that an analysis of syntax provides an automated (and therefore low-cost, rapid, and objective) method of identifying the concepts involved in a particular piece of code.

While in general there was a high level of similarity between the analysis of Python code and that of Java code, we did observe some clear differences between the two, and concluded that the outcome is dependent on the language. This is not in itself a problem, as the choice of language is known to contribute to intrinsic or extraneous cognitive load (depending on the viewpoint) for beginners. Studies concerning intuitiveness of programming languages have shown that Java performs poorly [81], which may be reflected by the results of our case studies especially in comparison to the results for Python.

It may be worth returning to the ability to *translate* or draw *maps* between different formal representations of the same entity, which

emerged from the learning objectives discussed in Section 4. This clearly characterises a broad category of concepts, including quite advanced topics such as translating from source to assembly-like code, but we can also identify small tasks that can be profitably approached at an early level. Here we mention just two examples related to the tasks in the above case studies:

(i) translation between *expression trees* and program expressions;
(ii) translation between *flow charts* and nested control structures, in particular *conditionals*.

In connection with **OP1** of case study 1 (section 6), for instance, a specific question might be:

TE1 Consider the following expression: a < 10 && s > 3
Which of the following is its corresponding expression tree?

[drawing of a few expression trees among which to choose]

Of course, students' understanding of expression trees could be assessed by following essentially the same process as in case study 1, but the important point to be addressed here is the relationship between different ways to represent the same thing. (Incidentally, the question in case study 1 may itself be seen as an instance of translation from an everyday-life description to its computational representation.) Moreover, if we can assume that the student has already a clear grasp of what expression trees mean, then task **TE1** also provides insight about their understanding, e.g., of operator precedence in textual expressions.

As a second example, in connection with **N1** of case study 2, we may ask:

TN1 Consider the following code segment:

```
int x = 1;
if (x < 3) {
  if (x < 1) {
    x = 2;
  }
  x = x + 5;
}
```

Which of the following is its corresponding flow chart?

[drawing of a few flow charts among which to choose]

Again, if we can assume that the student understands simple flow charts, task **TN1** gives us insight into their understanding of the control flow in a textual program.

Even though the decomposition of a task into lower-level components is feasible and has potential benefits, it is obvious that there are certain limitations to using the approach as a (sole) form of assessment. First, this decomposition process is very time-consuming and involves extensive discussions, and may therefore not be feasible in a normal class. Second, a collection of low-level assessments is not necessarily equivalent to the original question because it fails to assess the student's ability to synthesise the decomposed concepts.

However, this approach could be used for a selection of questions in a normal examination, to diagnose student misunderstandings, as scaffolding for learning tasks during a semester, or as part of mastery learning assessments.

9 CONCLUSION

This working group has surveyed the research literature to identify what concepts researchers and practitioners consider to be important in introductory programming courses, and examined the alignment between these concepts, the stated learning objectives for typical introductory programming courses, and the assessments used to evaluate student learning in such courses. We found a reasonably high degree of consistency between the concepts that were identified as important in the literature; we found consistent use of highly integrated assessments with similar frequency of concept components; but we also found that learning outcomes appeared to be highly variable.

An analysis of the grammar of Java allowed us to identify the language elements used in CS1 assessments and the relationships between those elements. The graph of the relationships 1 suggests a hierarchy of concepts, although it may be possible to progress in programming by understanding the elements present on particular paths through the graph, rather than needing to master all of the conceptual elements.

Finally, we demonstrated that it is possible to design assessments that focus on atomic concepts, rather than the more typical assessments that are highly interdependent. Appendix A provides an exemplar list of assessments that demonstrate how it might be possible to create assessments with limited interdependence.

This project is likely to be of particular interest to teachers and researchers involved in mastery learning, or those who are required to demonstrate tight coupling between learning outcomes and assessment tasks. It may also be of relevance to teachers and researchers interested in promoting a more positive mindset in the classroom, focusing on what students *do know* and what they *can achieve* during a typical CS1 course. A substantial body of research has focused on the use of exam questions and assessment items, but few publications link these assessment items to the intended learning outcomes of the courses in which they are used. We believe that this project will be of wide interest and has the potential to make a significant contribution to the field.

10 FUTURE WORK

Our findings suggest a number of possible avenues for future work. Following the working group, validation of the assessment items may present further opportunities for detailed rigorous analysis of student learning in CS1 courses. Here a Delphi process might be used to establish consensus among a small group of experts regarding the constituent parts of each concept. If the atomic questions are indeed valid and useful, it might be possible to tackle the problem of defining what makes the rainfall problem difficult [68, 73].

The learning outcomes for CS1 courses proved to be an interesting data source that warrants further investigation. In particular, to achieve a more thorough understanding of what the students are expected to learn, it may be helpful to examine the course syllabus specifications, which are usually published together with the learning outcomes themselves.

Although we have provided some exemplar questions in Java, it would be interesting to compare exemplar questions in Python to determine what differences in understanding are required for novices learning Python compared with those learning Java.

REFERENCES

[1] Alireza Ahadi and Raymond Lister. 2013. Geek Genes, Prior Knowledge, Stumbling Points and Learning Edge Momentum: Parts of the One Elephant?. In *Proceedings of the Ninth International Computing Education Research Conference (ICER '13)*. 123–128. https://doi.org/10.1145/2493394.2493416

[2] Deborah J. Armstrong. 2006. The Quarks of Object-oriented Development. *Commun. ACM* 49, 2 (February 2006), 123–128. https://doi.org/10.1145/1113034.1113040

[3] Association for Computing Machinery (ACM) and IEEE Computer Society, Joint Task Force on Computing Curricula. 2001. *Computing Curricula 2001: Computer Science.* ACM, New York, NY, USA.

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

[5] Mordechai Ben-Ari. 1998. Constructivism in Computer Science Education. In *Proceedings of the 29th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '98)*. 257–261. https://doi.org/10.1145/273133.274308

[6] Jens Bennedsen and Michael E. Caspersen. 2007. Failure Rates in Introductory Programming. *SIGCSE Bull.* 39, 2 (June 2007), 32–36. https://doi.org/10.1145/1272848.1272879

[7] Marc Berges, Andreas Mühling, and Peter Hubwieser. 2012. The Gap between Knowledge and Ability. In *Proceedings of the 12th Koli Calling International Conference on Computing Education Research (Koli Calling 2012)*. 126–134. https://doi.org/10.1145/2401796.2401812

[8] James H. Block and Robert B. Burns. 1976. Mastery Learning. *Review of Research in Education* 4, 1 (1976), 3–49. https://doi.org/10.3102/0091732X004001003

[9] Benjamin S. Bloom. 1968. Learning for Mastery. *Evaluation Comment* 1, 2 (1968).

[10] Benjamin S. Bloom. 1984. The 2 Sigma Problem: The Search for Methods of Group Instruction as Effective as One-to-One Tutoring. *Educational Researcher* 13, 6 (1984), 4–16. http://www.jstor.org/stable/1175554

[11] Richard Bornat, Saeed Dehnadi, and Simon. 2008. Mental Models, Consistency and Programming Aptitude. In *Proceedings of the Tenth Australasian Computing Education Conference (ACE '08)*. 53–61. http://dl.acm.org/citation.cfm?id=1379249.1379253

[12] Pauli Byckling and Jorma Sajaniemi. 2006. A Role-based Analysis Model for the Evaluation of Novices' Programming Knowledge Development. In *Proceedings of the Second International Computing Education Research Workshop (ICER '06)*. 85–96. https://doi.org/10.1145/1151588.1151602

[13] 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 SIGCSE Technical Symposium on Computing Science Education (SIGCSE '16)*. 364–369. https://doi.org/10.1145/2839509.2844559

[14] Dino Capovilla, Marc Berges, Andreas Mühling, and Peter Hubwieser. 2015. Handling Heterogeneity in Programming Courses for Freshmen. In *Proceedings of the International Conference on Learning and Teaching in Computing and Engineering (LaTiCE 2015)*. 197–203. https://doi.org/10.1109/LaTiCE.2015.18

[15] Yuliya Cherenkova, Daniel Zingaro, and Andrew Petersen. 2014. Identifying Challenging CS1 Concepts in a Large Problem Dataset. In *Proceedings of the 45th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '14)*. 695–700. https://doi.org/10.1145/2538862.2538966

[16] Liberal Arts Computer Science Consortium. 2007. A 2007 Model Curriculum for a Liberal Arts Degree in Computer Science. *J. Educ. Resour. Comput.* 7, 2, Article 2 (June 2007). https://doi.org/10.1145/1240200.1240202

[17] Albert T. Corbett and John R. Anderson. 1992. Student Modeling and Mastery Learning in a Computer-based Programming Tutor. In *Proceedings of the Intelligent Tutoring Systems: Second International Conference (ITS '92)*. 413–420. https://doi.org/10.1007/3-540-55606-0_49

[18] Malcolm Corney, Raymond Lister, and Donna Teague. 2011. Early Relational Reasoning and the Novice Programmer: Swapping as the Hello World of Relational Reasoning. In *Proceedings of the 13th Australasian Computing Education Conference (ACE '11)*. 95–104. http://crpit.com/confpapers/CRPITV114Corney.pdf

[19] Nell Dale. 2005. Content and Emphasis in CS1. *SIGCSE Bull.* 37, 4 (December 2005), 69–73. https://doi.org/10.1145/1113847.1113880

[20] Stephen Davies, Jennifer A. Polack-Wahl, and Karen Anewalt. 2011. A Snapshot of Current Practices in Teaching the Introductory Programming Sequence. In *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education (SIGCSE '11)*. 625–630. https://doi.org/10.1145/1953163.1953339

[21] Ton de Jong. 2010. Cognitive load theory, educational research, and instructional design: some food for thought. *Instructional Science* 38, 2 (01 Mar 2010), 105–134. https://doi.org/10.1007/s11251-009-9110-0

[22] Michael de Raadt, Richard Watson, and Mark Toleman. 2005. Textbooks: Under inspection. *Technical Report* – Department of Maths and Computing, University of Southern Queensland. Toowoomba, Queensland, Australia. https://eprints.usq.edu.au/167/1/TechReport_Draft_10.pdf – retrieved: September 2017.

[23] Benedict du Boulay. 1986. Some Difficulties of Learning to Program. *J. of Educational Comput. Research* 2, 1 (1986), 57–73.

[24] Anna Eckerdal, Robert McCartney, Jan Erik Moström, Mark Ratcliffe, and Carol Zander. 2006. Can Graduating Students Design Software Systems?. In *Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '06)*. 403–407. https://doi.org/10.1145/1121341.1121468

[25] Allison Elliott Tew and Mark Guzdial. 2010. Developing a Validated Assessment of Fundamental CS1 Concepts. In *Proceedings of the 41st SIGCSE Technical Symposium on Computer Science Education (SIGCSE '10)*. 97–101. https://doi.org/10.1145/1734263.1734297

[26] Allison Elliott Tew and Mark Guzdial. 2011. The FCS1: A Language Independent Assessment of CS1 Knowledge. In *Proceedings of the 42nd SIGCSE Technical Symposium on Computer Science Education (SIGCSE '11)*. 111–116. https://doi.org/10.1145/1953163.1953200

[27] Sophie Engle and Sami Rollins. 2013. Expert Code Review and Mastery Learning in a Software Development Course. *J. Comput. Sci. Coll.* 28, 4 (2013), 139–147.

[28] Heidi L. Eyre. 2007. Keller's Personalized System of Instruction: Was it a Fleeting Fancy or is there a Revival on the Horizon? *The Behavior Analyst Today* 8, 3 (2007), 317–324.

[29] Mark Guzdial and Briana Morrison. 2016. Growing Computer Science Education into a STEM Education Discipline. *Commun. ACM* 59, 11 (October 2016), 31–33. https://doi.org/10.1145/3000612

[30] John Hattie and Helen Timperley. 2007. The Power of Feedback. *Review of Educational Research* 77, 1 (March 2007), 81–112. https://doi.org/10.3102/003465430298487

[31] Matthew Hertz. 2010. What Do "CS1" and "CS2" Mean?: Investigating Differences in the Early Courses. In *Proceedings of the 41st SIGCSE Technical Symposium on Computer Science Education (SIGCSE '10)*. 199–203. https://doi.org/10.1145/1734263.1734335

[32] Cruz Izu, Amali Weerasinghe, and Cheryl Pope. 2016. A Study of Code Design Skills in Novice Programmers Using the SOLO Taxonomy. In *Proceedings of the 12th International Computing Education Research Conference (ICER '16)*. 251–259. https://doi.org/10.1145/2960310.2960324

[33] Fred. S. Keller. 1968. Goodbye, Teacher ... *Journal of Applied Behavior Analysis* 1 (1968), 79–89.

[34] Chen-Lin C. Kulik, James A. Kulik, and Robert L. Bangert-Drowns. 1990. Effectiveness of Mastery Learning Programs: A Meta-Analysis. *Review of Educational Research* 60, 2 (1990), 265–299. https://doi.org/10.3102/00346543060002265

[35] Amruth N. Kumar. 1996. Fork Diagrams for Teaching Selection in CS I. In *Proceedings of the 27th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '96)*. 348–352. https://doi.org/10.1145/236452.236575

[36] Noel LeJeune. 2010. Contract Grading with Mastery Learning in CS 1. *J. Comput. Sci. Coll.* 26, 2 (December 2010), 149–156. http://dl.acm.org/citation.cfm?id=1858583.1858604

[37] Colleen M. Lewis. 2014. Exploring Variation in Students' Correct Traces of Linear Recursion. In *Proceedings of the Tenth International Computing Education Research Conference (ICER '14)*. 67–74. https://doi.org/10.1145/2632320.2632355

[38] 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 *2007 ITiCSE Working Group Reports (ITiCSE-WGR '04)*. 119–150. https://doi.org/10.1145/1044550.1041673

[39] Raymond Lister, Beth Simon, Errol Thompson, Jacqueline L. Whalley, and Christine Prasad. 2006. Not Seeing the Forest for the Trees: Novice Programmers and the SOLO Taxonomy. In *Proceedings of the 11th Conference on Innovation and Technology in Computer Science Education (ITICSE '06)*. 118–122. https://doi.org/10.1145/1140124.1140157

[40] 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 Computing Education Research Workshop (ICER '08)*. 101–112. https://doi.org/10.1145/1404520.1404531

[41] Andrew Luxton-Reilly. 2016. Learning to Program is Easy. In *Proceedings of the 21st Conference on Innovation and Technology in Computer Science Education (ITiCSE '16)*. 284–289. https://doi.org/10.1145/2899415.2899432

[42] Andrew Luxton-Reilly and Andrew Petersen. 2017. The Compound Nature of Novice Programming Assessments. In *Proceedings of the 19th Australasian Computing Education Conference (ACE '17)*. 26–35. https://doi.org/10.1145/3013499.3013500

[43] Linxiao Ma, John Ferguson, Marc Roper, and Murray Wood. 2007. Investigating the Viability of Mental Models Held by Novice Programmers. In *Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '07)*. 499–503. https://doi.org/10.1145/1227310.1227481

[44] Philipp Mayring. 2014. *Qualitative Content Analysis: Theoretical Foundation, Basic Procedures and Software Solution*. Klagenfurt.

[45] Brendan McCane, Claudia Ott, Nick Meek, and Anthony Robins. 2017. Mastery Learning in Introductory Programming. In *Proceedings of the 19th Australasian Computing Education Conference (ACE '17)*. 1–10. https://doi.org/10.1145/3013499.3013501

[46] Robert McCartney, Jonas Boustedt, Anna Eckerdal, Kate Sanders, and Carol Zander. 2013. Can First-year Students Program Yet?: A Study Revisited. In *Proceedings of the Ninth International Computing Education Research Conference (ICER '13)*. 91–98. https://doi.org/10.1145/2493394.2493412

[47] Michael McCracken, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, Ian Utting, and Tadeusz Wilusz. 2001. A Multi-national, Multi-institutional Study of Assessment of Programming Skills of First-year CS Students. *SIGCSE Bull.* 33, 4 (December 2001), 125–180. https://doi.org/10.1145/572139.572181

[48] Iain Milne and Glenn Rowe. 2002. Difficulties in Learning and Teaching Programming—Views of Students and Tutors. *Education and Information Technologies* 7, 1 (March 2002), 55–66. https://doi.org/10.1023/A:1015362608943

[49] Luiz Augusto de Macêdo Morais, Jorge C. A. Figueiredo, and Dalton D. S. Guerrero. 2014. Students Satisfaction with Mastery Learning in an Introductory Programming Course. In *III Congresso Brasileiro de Informática na Educação (CBIE 2014)*. 1048–1052. https://doi.org/10.5753/cbie.sbie.2014.1048

[50] Briana B. Morrison, Lauren E. Margulieux, and Mark Guzdial. 2015. Subgoals, Context, and Worked Examples in Learning Computing Problem Solving. In *Proceedings of the 11th Annual International Computing Education Research Conference (ICER '15)*. 21–29. https://doi.org/10.1145/2787622.2787733

[51] Andreas Mühling, Alexander Ruf, and Peter Hubwieser. 2015. Design and First Results of a Psychometric Test for Measuring Basic Programming Abilities. In *Proceedings of the Workshop in Primary and Secondary Computing Education (WiPSCE '15)*. 2–10. https://doi.org/10.1145/2818314.2818320

[52] Pavol Návrat. 1994. Hierarchies of Programming Concepts: Abstraction, Generality, and Beyond. *SIGCSE Bull.* 26, 3 (September 1994), 17–21. https://doi.org/10.1145/187387.187397

[53] Miranda C. Parker, Mark Guzdial, and Shelly Engleman. 2016. Replication, Validation, and Use of a Language Independent CS1 Knowledge Assessment. In *Proceedings of the 12th International Computing Education Research Conference (ICER '16)*. 93–101. https://doi.org/10.1145/2960310.2960316

[54] Terence Parr and Sam Harwell. 2012–2016. Java 1.7 grammar for ANTLR v4. https://github.com/antlr/grammars-v4/blob/master/java/Java.g4. (2012–2016).

[55] Elizabeth Patitsas, Jesse Berlin, Michelle Craig, and Steve Easterbrook. 2016. Evidence That Computer Science Grades Are Not Bimodal. In *Proceedings of the 12th International Computing Education Research Conference (ICER '16)*. 113–121. https://doi.org/10.1145/2960310.2960312

[56] Michela Pedroni and Bertrand Meyer. 2010. Object-Oriented Modeling of Object-Oriented Concepts.. In *Proceedings of the Fourth International Conference on Informatics in Secondary Schools - Evolution and Perspectives (ISSEP 2010)*. Springer, 155–169.

[57] Andrew Petersen, Michelle Craig, Jennifer Campbell, and Anya Tafliovich. 2016. Revisiting Why Students Drop CS1. In *Proceedings of the 16th Koli Calling International Conference on Computing Education Research (Koli Calling 2016)*. 71–80. https://doi.org/10.1145/2999541.2999552

[58] Andrew Petersen, Michelle Craig, and Daniel Zingaro. 2011. Reviewing CS1 Exam Question Content. In *Proceedings of the 42nd SIGCSE Technical Symposium on Computer Science Education (SIGCSE '11)*. 631–636. https://doi.org/10.1145/1953163.1953340

[59] Jean Piaget. 1976. *Piaget and His School: A Reader in Developmental Psychology*. Springer Berlin Heidelberg, Germany.

[60] Viera K. Proulx. 2000. Programming Patterns and Design Patterns in the Introductory Computer Science Course. In *Proceedings of the 31st SIGCSE Technical Symposium on Computer Science Education (SIGCSE '00)*. 80–84. https://doi.org/10.1145/330908.331819

[61] Anthony Robins. 2010. Learning Edge Momentum: A New Account of Outcomes in CS1. *Computer Science Education* 20, 1 (March 2010), 37–71. https://doi.org/10.1080/08993401003612167

[62] Anthony Robins, Janet Rountree, and Nathan Rountree. 2003. Learning and Teaching Programming: A Review and Discussion. *Computer Science Education* 13, 2 (June 2003), 137–172. https://doi.org/10.1076/csed.13.2.137.14200

[63] Jorma Sajaniemi, Mordechai Ben-Ari, Pauli Byckling, Petri Gerdt, and Yevgeniya Kulikova. 2006. Roles of Variables in Three Programming Paradigms. *Computer Science Education* 16, 4 (December 2006), 261–279. https://doi.org/10.1080/08993400600874584

[64] 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 *2013 ITiCSE Working Group Reports (ITiCSE-WGR '13)*. 33–52. https://doi.org/10.1145/2543882.2543885

[65] Kate Sanders and Lynda Thomas. 2007. Checklists for Grading Object-oriented CS1 Programs: Concepts and Misconceptions. In *Proceedings of the 12th Conference on Innovation and Technology in Computer Science Education (ITiCSE '07)*. 166–170. https://doi.org/10.1145/1268784.1268834

[66] Tamarisk Lurlyn Scholtz and Ian Sanders. 2010. Mental Models of Recursion: Investigating Students' Understanding of Recursion. In *Proceedings of the 15th Conference on Innovation and Technology in Computer Science Education (ITiCSE*

'10). 103–107. https://doi.org/10.1145/1822090.1822120

[67] Carsten Schulte and Jens Bennedsen. 2006. What Do Teachers Teach in Introductory Programming?. In *Proceedings of the Second International Computing Education Research Workshop (ICER '06)*. 17–28. https://doi.org/10.1145/1151588.1151593

[68] Otto Seppälä, Petri Ihantola, Essi Isohanni, Juha Sorva, and Arto Vihavainen. 2015. Do We Know How Difficult the Rainfall Problem Is?. In *Proceedings of the 15th Koli Calling Conference on Computing Education Research (Koli Calling 2015)*. 87–96. https://doi.org/10.1145/2828959.2828963

[69] Judy Sheard, Angela Carbone, Raymond Lister, Beth Simon, Errol Thompson, and Jacqueline L. Whalley. 2008. Going SOLO to Assess Novice Programmers. In *Proceedings of the 13th Conference on Innovation and Technology in Computer Science Education (ITiCSE '08)*. 209–213. https://doi.org/10.1145/1384271.1384328

[70] Judy Sheard, Simon, Angela Carbone, Donald Chinn, Tony Clear, Malcolm Corney, Daryl D'Souza, Joel Fenwick, James Harland, Mikko-Jussi Laakso, and Donna Teague. 2013. How Difficult Are Exams?: A Framework for Assessing the Complexity of Introductory Programming Exams. In *Proceedings of the 15th Australasian Computing Education Conference (ACE '13)*. 145–154. http://dl.acm.org/citation.cfm?id=2667199.2667215

[71] Shuhaida Shuhidan, Margaret Hamilton, and Daryl D'Souza. 2009. A Taxonomic Study of Novice Programming Summative Assessment. In *Proceedings of the 11th Australasian Computing Education Conference (ACE '09)*. 147–156. http://dl.acm.org/citation.cfm?id=1862712.1862734

[72] Robert M. Siegfried, Jason P. Siegfried, and Gina Alexandro. 2016. A Longitudinal Analysis of the Reid List of First Programming Languages. *Information Systems Education Journal* 14, 6 (November 2016), 47–54. http://isedj.org/2016-14/

[73] Simon. 2013. Soloway's Rainfall Problem Has Become Harder. In *Proceedings of the 2013 Conference on Learning and Teaching in Computing and Engineering*. 130–135. https://doi.org/10.1109/LaTiCE.2013.44

[74] Simon, Donald Chinn, Michael de Raadt, Anne Philpott, Judy Sheard, Mikko-Jussi Laakso, Daryl D'Souza, James Skene, Angela Carbone, Tony Clear, Raymond Lister, and Geoff Warburton. 2012. Introductory Programming: Examining the Exams. In *Proceedings of the 14th Australasian Computing Education Conference (ACE '12)*. 61–70. http://dl.acm.org/citation.cfm?id=2483716.2483724

[75] 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 21st Conference on Innovation and Technology in Computer Science Education (ITiCSE '16)*. 154–159. https://doi.org/10.1145/2899415.2899473

[76] 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 12th International Computing Education Research Conference (ICER '16)*. 103–111. https://doi.org/10.1145/2960310.2960337

[77] Beth Simon, Tzu-Yi Chen, Gary Lewandowski, Robert McCartney, and Kate Sanders. 2006. Commonsense Computing: What Students Know before We Teach (Episode 1: Sorting). In *Proceedings of the Second International Computing Education Research Workshop (ICER '06)*. 29–40. https://doi.org/10.1145/1151588.1151594

[78] Ben Skudder and Andrew Luxton-Reilly. 2014. Worked Examples in Computer Science. In *Proceedings of the 16th Australasian Computing Education Conference (ACE '14)*. 59–64. http://dl.acm.org/citation.cfm?id=2667490.2667497

[79] Juha Sorva. 2012. *Visual Program Simulation in Introductory Programming Education*. Ph.D. Dissertation. Aalto University.

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

[81] Andreas Stefik and Susanna Siebert. 2013. An Empirical Investigation into Programming Language Syntax. *ACM Transactions on Computing Education* 13, 4 (2013), 19:1. https://doi.org/10.1145/2534973

[82] John Sweller. 1988. Cognitive Load during Problem Solving: Effects on Learning. *Cognitive Science* 12, 2 (1988), 257–285. https://doi.org/10.1207/s15516709cog1202_4

[83] John Sweller. 1994. Cognitive Load Theory, Learning Difficulty, and Instructional Design. *Learning and Instruction* 4 (1994), 295–312.

[84] John Sweller and Paul Chandler. 1994. Why Some Material Is Difficult to Learn. *Cognition and Instruction* 12, 3 (1994), 185–233. https://doi.org/10.1207/s1532690xci1203_1

[85] Cynthia Taylor, Daniel Zingaro, Leo Porter, Kevin Webb, Cynthia Bailey Lee, and Mike Clancy. 2014. Computer Science Concept Inventories: Past and Future. *Computer Science Education* 24, 4 (October 2014), 253–276. https://doi.org/10.1080/08993408.2014.970779

[86] Donna Teague and Raymond Lister. 2014. Longitudinal Think Aloud Study of a Novice Programmer. In *Proceedings of the 16th Australasian Computing Education Conference (ACE '14)*. 41–50. http://dl.acm.org/citation.cfm?id=2667490.2667495

[87] Mark Urban-Lurain and Donald J. Weinshank. 1999. "I Do and I Understand": Mastery Model Learning for a Large Non-major Course. In *Proceedings of the 30th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '99)*. 150–154. https://doi.org/10.1145/299649.299738

[88] Anne Venables, Grace Tan, and Raymond Lister. 2009. A Closer Look at Tracing, Explaining and Code Writing Skills in the Novice Programmer. In *Proceedings of the Fifth International Computing Education Research Workshop (ICER '09)*. 117–128. https://doi.org/10.1145/1584322.1584336

[89] Christopher Watson and Frederick W.B. Li. 2014. Failure Rates in Introductory Programming Revisited. In *Proceedings of the 19th Conference on Innovation and Technology in Computer Science Education (ITiCSE '14)*. 39–44. https://doi.org/10.1145/2591708.2591749

[90] Jacqueline Whalley and Nadia Kasto. 2014. How Difficult Are Novice Code Writing Tasks?: A Software Metrics Approach. In *Proceedings of the 16th Australasian Computing Education Conference (ACE '14)*. 105–112. http://dl.acm.org/citation.cfm?id=2667490.2667503

[91] Jacqueline L. Whalley and Raymond Lister. 2009. The BRACElet 2009.1 (Wellington) Specification. In *Proceedings of the 11th Australasian Computing Education Conference (ACE '09)*. 9–18. http://dl.acm.org/citation.cfm?id=1862712.1862717

[92] 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 Eighth Australasian Computing Education Conference (ACE '06)*. 243–252. http://dl.acm.org/citation.cfm?id=1151869.1151901

[93] Mitsuo Yamamoto, Takayuki Sekiya, Kazumasa Mori, and Kazunori Yamaguchi. 2012. Skill Hierarchy Revised by SEM and Additional Skills. In *Proceedings of the International Conference on Information Technology Based Higher Education and Training (ITHET 2012)*. 1–8.

[94] Daniel Zingaro, Andrew Petersen, and Michelle Craig. 2012. Stepping up to Integrative Questions on CS1 Exams. In *Proceedings of the 43rd SIGCSE Technical Symposium on Computer Science Education (SIGCSE '12)*. 253–258. https://doi.org/10.1145/2157136.2157215

APPENDIX A   ASSESSMENT TASKS FOR JAVA

The following list contains all the elements appearing in Figure 1 that relate to procedural programming, together with assessment tasks that show a decomposition of each element into its basic constituent parts.

A.1   expression

There are many different operators and literals that can be used to form an expression. The major point that learners need to know in each case is that an expression results in a value and that this value is – often – computed by evaluating terms using operators based on certain precedence rules. This applies to expressions in their most basic forms. More complex expressions that rely on method calls, for example, should be tested accordingly by using appropriate items for the concepts that go beyond the basic expressions. Here we give only items that belong to boolean literals (BL1) and the three boolean operators (BO1-BO3) including their precedence rules (BP1). Additionally, although it is not considered an expression in Java, we include items for the concept of assignment (A1-A4), which does not appear as a separate entity in the dependency graph.

BL1 List all of the boolean values

BO1 Give the results of the following expressions:
true && true, true && false, false && true, false && false

BO2 Give the results of the following expressions:
true || true, true || false, false || true, false || false

BO3 Give the results of the following expressions:
!true, !false

BP1 In the expression *!a && b || c*, which of the operators (!, || , &&) is evaluated first and which is evaluated second?

A1 What type must variable *x* have for the code segment *x = 1;* to work successfully?

A2 What is the value of the variable *x* after the code segment *x = 1;* has been run?

A3 Are the two segments *x = 1;* and *1 = x;* equivalent?

A4 What is the value of *a* after executing the following code segments where *a* and *b* are variables of type int?

```
a = 1;
b = a + 2;
a = 3;
```

A.2 identifier

I1 Which of the following code elements are identifiers and which are literals?

gender, 3.0, "giraffe", 5, x

I2 Can a variable hold/represent/refer to different values at different times during the execution of a program?

I3 If a variable named *x* contains the value 5, what will *x + 7* be?

A.3 literal

L1 Same as I1

A.4 floatingPointLiteral & integerLiteral & ... & stringLiteral

All literals are included in a broader "typeLiteral" item.

TL1 Which of the following literals is of type [...] ?

'x', 3.0, "giraffe", 5, true, null

A.5 typeType

T1 Does every value in Java have a type?

A.6 primitiveType

PT1 Which of the following literals is of a primitive type?

'x', 3.0, "giraffe", 5, true, 1,2,3

PT2 If the following Java code is valid, can *x* be of a primitive type?

```
x.size();
```

A.7 classOrInterfaceType

CIT1 Which of the following literals is not of a primitive type?

'x', 3.0, "giraffe", 5, true, 1,2,3

CIT2 If the following Java Code is valid, must *x* be of a primitive Type?

```
x.size();
```

A.8 localVariableDeclaration

LV1 Same as FS2

A.9 variableDeclarator

VD1 Can the value of a variable change during the execution of a piece of code?

Can the value of a variable stay the same during the execution of a piece of code?

VD2 Is the following code valid?

```
int x;
int x;
```

A.10 variableInitializer

VI1 Is the code segment *int x = 1;* equivalent to the following?

```
int x;
x = 1;
```

A.11 arrayInitializer

AI1 How many elements does *x* have after the following code is executed?

```
int[] x = {2,1,3};
```

AI2 What element is at the the first position of *x* after the following code is executed?

```
int[] x = {3,2,1};
```

AI2 Is the code segment *int[] x = 1,2,3;* equivalent to the following?

```
int[] x;
x = {1,2,3};
```

A.12 statement

S1 Which of the following code segments denotes a statement?

```
x + 1
x + y;
```

A.13 ifStatement

IS1 What is the value of *x* after the execution of the following program segment?

```
int x = 1;
if (x > 2)
 x = 2;
```

IS2 What is the value of *x* after the execution of the following program segment?

```
int x = 1;
if (x > 2)
 x = 2;
else
 x = 0;
```

A.14 forStatement

FS1 What is the value of *x* after the execution of the following program segment?

```
int x = 1;
for (int i = 0; i < 5; i++) {
 x = i;
}
```

FS2 Is the following code correct?

```
int x = 1;
for (int i = 0; i < 5; i++) {
 x = x + 1;
}
x = i;
```

A.15 whileStatement

WS1 Assuming that a variable x of type int has been declared, which of the following would be valid beginnings for a while loop?

while (x), while (x < 10), while (x + 3), while (x == 4)

WS2 What is the value of x after the execution of the following program segment?

```
int x = 1;
while (x < 2) {
 x = 10;
 x = x - 5;
}
```

WS3 True or false: it is possible for a while loop never to terminate.

A.16 switchStatement & switchLabel

SS1 What value must x hold to ensure that the value of y is 100 after the following code segment is executed?

```
int y = 0;
switch (x) {
 case 1: y = 10; break;
    case 2: y = 100; break;
    case 3: y = 1000;
}
```

SS2 What is the value of y after the execution of the following program segment?

```
int x = 1;
int y = 0;
switch (x) {
 case 1: y = 10;
    case 2: y = 100;
    case 3: y = 1000;
}
```

A.17 breakStatement

BS1 What is the value of x after the execution of the following program segment?

```
int x = 1;
while (x > 0) {
   break;
   x = 2;
}
```

A.18 continueStatement

CS1 What is the value of x after the execution of the following program segment?

```
int x = 1;
for (int i = 0; i < 10; i++) {
   continue;
   x = 2;
}
```

A.19 block

The element "block" encompasses the ideas of sequences (S1, S2), simple blocks (B1), and nested blocks (N1).

S1 Is the code segment *int x = 1; int y = 2;* equivalent to the following code?

```
int x;
x = 1;
int y;
y = 2;
```

S2 Will the individual statements in the following code segment always be executed in the order that they are written?

```
int x = 1;
int y = 3;
x = y + 4;
```

B1 What is the value of x after the execution of each of the following program segments?

(1)
```
int x = 1;
if (x > 2) {
   x = 2;
   x = x + 5;
}
```

(2)
```
int x = 1;
if (x > 2) {
 x = 2;
}
x = x + 5;
```

N1 What is the value of x after the execution of the following program segment?

```
int x = 1;
if (x < 3) {
   if (x < 1) {
    x = 2;
   }
   x = x + 5;
}
```

A.20 arguments & typeArguments & expressionList

A1 What value is used as the argument of the method called in the code segment below?

```
String s1 = "Mary";
String s2 = "Marianne";
int x = s1.compareTo(s2);
```

TA1 What is the type of the argument of the method called in the code segment below?

```
String s1 = "Mary";
String s2 = "Marianne";
int x = s1.compareTo(s2);
```

EL1 How many expressions are there in the statement below?

```
s.indexOf('a', 2);
```

A.21 methodCall

MC1 What is the name of the method that is called in the code segment below?

```
String s = "Mary";
int x = s.length();
```