Teaching and Assessing Programming Language Tracing

Greg L. Nelson

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2021

Reading Committee:

Amy J. Ko, Chair

Rastislav Bodik

Jeffrey Heer

Program Authorized to Offer Degree:
Computer Science and Engineering

University of Washington

Abstract

Teaching and Assessing Programming Language Tracing

Greg L. Nelson

Chair of the Supervisory Committee:
Professor Amy J. Ko
The Information School

Learning to write programs is hard, but many fail to even learn basic program reading skills, such as mentally tracing a program to predict its behavior. This dissertation argues **a new theory of programming language knowledge that includes mappings from syntax to semantics and their nested combinations can serve as the basis for more granular tools for learning and more precise assessments of that knowledge.** First, I created a new theory of basic programming language knowledge as knowing the mapping from token-level syntax to semantics encoded in a PL interpreter's execution paths, and nested combinations of those paths. I proposed this knowledge can be learned by humans via causal inference, drawing on recent research in psychology. I used this theory to design a new reading-first spiral curriculum approach for learning programming that directly shows and teaches program tracing, without program writing. I implemented this curriculum in PLTutor, an interactive textbook; in a comparative study versus an interactive writing tutorial (Codecademy), I found initial evidence of improved learning gains on the SCS1, with average learning gains of PLTutor 60% higher than Codecademy. PLTutor students also did not fail their midterm, versus >10% who failed in the Codecademy group. Second, I created a new formative assessment for program tracing, with precise questions systematically generated to cover that knowledge, including nested combinations. My evaluation with 31 people found per question error and guessing rates generally within desirable thresholds for educational assessment, and a ~70% success rate for targeting precise feedback for learning. Finally, I created differentiated assessments,

a new genre of assessment question designed to diagnose learner issues more precisely for an advanced topic versus prerequisite program tracing skills. I led a collaboration to 1) empirically show existing advanced topic assessments depend on advanced and prerequisite knowledge, and have difficulty precisely differentiating between the two, and 2) create example questions and design guidelines for more precise differentiated assessments. Together, this thesis advances teaching and assessment for basic program tracing skills, and concretely raises the possibility that other skills in computing might be taught better by asking what a theory of the skill might be, how to teach at a useful level of granularity, and how to more precisely assess that skill both directly and as a dependency when assessing advanced skills. This thesis concludes with a vision of programming education that includes reading and learning from great works of code and the meaningful human stories of their creation.

TABLE OF CONTENTS

LIST OF FIGURES

Figure Number Page

LIST OF TABLES

ACKNOWLEDGMENTS

I thank my parents, Steven Lloyd Nelson and Nancy Mary Nelson (maiden name Thanghe and mother's family name Curwick), for the gift of my life and the security of a place to call home. They have given me the best love that parents can give, and always encouraged me to live the life I want to have, rather than meet some expectations. Their tireless work over thirty years in the paper industry and dental hygiene has provided a home and a safety net for me that supported me to take risks and be a part of something beyond my immediate needs.

I thank my advisor, Amy Ko, for supporting me as a researcher and as a person, being understanding and accommodating of my disabilities, and seeing my talents and valuing creativity. I am also thankful for the Computer Science department, particularly Lisa Singh, Mark Maloof, and Richard Squier, who helped me have my first adventures in computing in a warm, personal, and supportive environment.

I thank my committee members for their patience, encouragement, and feedback. Thank you Jeff, for helping in my early formation as a researcher, and welcoming times and collaborations in the IDL. Thank you Jason, for being available and welcoming over the years in helping me better understand the learning sciences. Thank you Ras, for your open-mindedness to see the value in computing education within computer science, and discussion on automating analyses for future work. Thank you Elizabeth, for your help and thoughts on study design while also tampering my ambitions to be more realistic.

I also offer thanks to all of the lab mates, collaborators, students, faculty, staff, mentors, draft-readers, and friends that have made this work possible including Benjamin Xie, Shana Hutchison, Eric Whitmire, Pavel Panchecka, Alannah Oleson, Eric Butler, Aaron Bauer, Kyle Thayer, Matt Kay, Morgan Dixon, Danielle Bragg, Yun-en Li, Jonathan Bragg, Kit Kuksenock, Matt Davidson,

Dharma Dailey, Ahmer Arif, Lindsay Michimoto, Annie Ross, Chenglong Wang, Eric Hekler, Matt Davidson, Andrew Hu, Alex Hui Tan, Leanne Hwa, Mohit Jain, Filip Strömbäck, Ari Korhonen, Marjahan Begum, Ben Blamey, Karen H. Jin, Violetta Lonati, Bonnie MacKellar, Mattia Monga, Steve Tanimoto, Alan Borning, Daniela Rosner, Richard Ladner, Gaetano Borriello, BJ Burg, Rahul Banerjee, Arvind Satyanarayan, Dominik Moritz, Andrea Verdan, Rob Thompson, Luheng He, first year lab mates, board games crew, Edward Zhang, Xuan Luo, Griffin Nicoll, Facebook and Oculus folks, Elizabeth Patitsas, Karl Koscher, Conrad Nied, Amanda Swearngin, Stefania Druga, Mara Kirdani-Ryan, Andrew Hu, Jinna Lei, Saba Kawas, Yim Register, Nicole Cederblom, Travis Kriplean, Michael Toomim, Paul Pham, Cynthia Matuszek, Michael Lee, Adam Smith, Kathleen Tuite, Cynthia Bennet, Daniel Perelman, Brett Wortzman, Zorah Fung, Tamara Denning, Daniel Epstein, Josh Pollock, Mina Tari, Calvin Apodaca, Mayank Goel, Lisa Elkin, Alex Polozov, Hanchuan Li, Elliot Saba, Lilian de Greef, Dhruv Jain, W.E. King, Alex Mariakakis, Edward Wang, Divye Jain, Elise de Gorough Goede, Iman Yeckehzaare, Qiao Zhang, Cecilia Aragon, Eric Klavins, Bill Howe, James Fogarty, Axel Roesler, Dan Weld, Joshua Smith, Daniela Rosner, Yoshi Kohno, Juha Sorva, Lauri Malmi, Otto Seppälä, Benedict du Boulay, Mark Guzdial, Barbara Ericson, Shriram Krishnamurthi, Kathi Fisler, Briana Morrison, Lionel Deimel, Judy Sheard, Marian Petre, Colleen Lewis, Jun Kato, Jeffrey Stylos, Leigh Ann DeLyser, Benjamin R. Shapiro, Alexander Repenning, Yoshiki Ohshima, Alan Kay, James Prather, Brett Becker, Monica McGill, Peter Brusilovsky, Allison Elliott Tew, Shih-wen Huang, Harrison Kwik, William Menten-Weil, Matthew Conlen, James T. Lamiell, Michael Somos, Paul Kainen, Andrew Vogt, Nick Kiritz, Jevin West, Diana Stanescu, Tanmay Sinha, Samantha Finkelstein, Caroline Pitt, Meron Solomon, Chandrakana Nandi, Sam Saarinen and Douglas Engelbart. You all have helped me navigate through the tides of the Ph.D. program and inspired my work, helped me see things in new ways, laughed and smiled, given me the gift of criticism, taken time to share your passions and expertise, and helped me grow in one way or another. Thank you all, I could not have done it without you :)

Lastly, I want to express thanks and gratitude for the many systems I depend on and emerged

with. In this dissertation I use the word I in statements like “I did this work” as a more proximal cause, but I am also inseparable from the strands of history, ideas, support, mentorship, farmers, builders, workers, and ecosystems that form a greater whole.

Chapter 1

INTRODUCTION

Learning programming is hard and requires both learning program writing and program reading skills. Program writing requires many complex skills, including planning, program design, and problem domain knowledge [Loksa et al., 2016, Loksa and Ko, 2016, Robins et al., 2003]. The most basic level of program reading skill is *program tracing*, which is knowing how programs execute and being able to mentally execute a program given concrete input values [Mayrhauser and Vans, 1995, Lopez et al., 2008, Lister et al., 2009, Venables et al., 2009, Busjahn and Schulte, 2013]. Unfortunately, even after taking a CS1 undergraduate course, learners struggle to master even basic program tracing skills: two large multinational studies show more than 60% of students incorrectly answer questions about the execution of basic programs [Lister et al., 2004b, McCracken et al., 2001]. These reading difficulties appear to persist for more than 30% of 3rd and 4th year students and correlate with worse final exam scores [Valstar et al., 2019, Fisler et al., 2017].

Teachers and researchers have attempted to address these difficulties in seemingly diverse ways, creating [Kemeny et al., 1968, Maloney et al., 2010, Pane, 2002, Papert, 1971, Pausch et al., 2000, Ingalls, 2020] or changing programming languages [Clark et al., 1998, Lecarme, 1974, De Raadt et al., 2002, Ralston, 1971, Roberts, 2004], tools, and pedagogy. Visualization tools make hidden state more explicit, showing execution at the line level [Barr et al., 1975b, Yehezkel, 2003, Virtanen et al., 2005, Guo, 2013a] to the sub-expression level [Lieberman, 1984, Moreno et al., 2004, Sorva and Sirkia, 2010]. Other tools have changed the domain of application, to improve visibility of program state and motivation [Lee and Ko, 2015, Maloney et al., 2010, Pausch et al., 2000]. Other tools increase writing [PracticeIt, 2016, CodingBat, 2016] and tracing practice via online courseware [Karavirta et al., 2015]. People have tried to evolve pedagogy by reordering [Van Merrienboer and Krammer, 1987, Turbak et al., 1999, Downey and Stein, 2006, Palumbo,

1990, Sorva and Seppälä, 2014] and changing what is taught [Campbell and Bolker, 2002, Luxton-Reilly, 2016, Reges, 2006]. People have tried deeper changes: presenting knowledge via worked examples [Morrison et al., 2015], teaching problem solving more explicitly [Palumbo, 1990, Loksa et al., 2016], teaching program design more [Felleisen et al., 2001], and even larger changes such as a discovery pedagogy [Littlefield et al., 1988].

While these attempts over the past seventy years appear diverse, they all have been designed and evaluated within a *writing*-focused pedagogical context. When people design programming classes or online tutorials, nearly everyone focuses on writing practice immediately with minimal to no reading practice. Typically, learners receive instruction on a programming language (PL) construct's syntax and semantics, practice by writing code, then advance to the next construct; this spiral writing approach goes back to the late 1970s at least [Shneiderman, 1977, Lemos, 1979]).

In contrast, when we learn to read and write natural language, we learn to read first and our reading skills tend to stay ahead of our writing skills; yet in computing education little prior work has explored a *reading*-first pedagogy, teaching *program tracing*—how static code causes dynamic computer behavior—*before* teaching learners to write code. Prior work proposes some approaches and curricular ordering [Deimel and Moffat, 1982, Kimura, 1979, Shneiderman, 1977, Ben-Ari, 2001, Sorva, 2013a] but lacks implementation, evaluation, and, more fundamentally, a detailed theory of basic programming language knowledge to guide design for instruction and precise assessments.

To create a reading-first curriculum and instruction, and program tracing assessments, we might find it useful to start from a theory of what basic programming language (PL) knowledge is and how people can learn that knowledge [Niss, 2005, Malmi et al., 2019, Nelson and Ko, 2018]. That theory can help design curriculum and instruction with sufficient *granularity*: showing the parts of the PL with enough detail to reliably learn each one and put them together, to avoid overwhelming the learner or even not showing parts of that knowledge, which forces the learner to infer it. That theory can also divide the knowledge into parts, to help design more *precise* assessments that aid learning by providing targeted feedback.

In this thesis I demonstrate **a new theory of programming language knowledge that includes**

mappings from syntax to semantics and their nested combinations can serve as the basis for more granular tools for learning and more precise assessments of that knowledge.

This thesis is organized in the following way. In Chapter 2 I review the gaps in theory of programming language knowledge, granular tools for learning, and precise assessments.

In Chapter 3 I develop a **theory of basic programming language knowledge that includes the mapping from token-level syntax to semantics and machine state**, as encoded in a PL interpreter's execution paths. For example, in `var x=1;` the `x=` assigns a value to `x` in the variable assignment table in the machine state. The key motivation was to decompose that PL knowledge into small granular parts, to enable learning via human causal inference. See Section 3.1 for more detail.

I then use that theory as the basis for creating a new reading-first spiral curriculum approach and implement that curriculum as a self-contained interactive textbook: PLTutor. PLTutor is a new **more granular tool for learning** for teaching that knowledge, enabled by *learning steps*, a teaching layer of abstraction on top of program visualization systems, which supports flexible teaching that mixes conceptual instruction, program visualization, and scaffolded assessments. For more details, see Section 3.2.1 for instructional design and Section 3.2.2 for curriculum design and flexible teaching granularity.

In Chapter 4 I extend that theory to include **nested combinations** of syntax to semantics mappings, to create a **more precise assessment** of program tracing skills. For example, one nested combination is a nested `if` statement where the outer `if` condition is true and the inner `if` condition is true; another separate combination covers the outer condition is false and inner condition is true. I then create a systematic test of those nested combinations, by generating many questions that each tested a separate part of that knowledge. I drew on Kane's educational assessment validity framework to make an argument for validity for using the test to target feedback for learning, supported by an empirical study of that specific use.

In Chapter 5 I extend my theory of programming language knowledge to include relationships with advanced skills, identifying the problem that assessments of advanced topics aren't precise enough to distinguish between learner issues with an advanced skill versus its prerequisites. Based

on that theoretical insight, I co-led a collaboration to 1) establish the problem exists via an empirical qualitative study of existing assessments with respect to prerequisite program tracing skills (see Section 5.4), and 2) create example questions and design guidelines for *differentiated assessments*, designed to distinguish between advanced and prerequisite program tracing skills (see Section 5.5).

I conclude in Chapter 6 with a summary of contributions, discussion, and future work. The Appendices include the reading-first spiral curriculum from Chapter 3 and the questions analyzed and modified in Chapter 5.

Chapter 2

RELATED WORK

This chapter starts with background from learning sciences and educational assessment research. I then note gaps in work in computing education research in Section 2.3 and Section 2.4, and the corresponding parts of this thesis filling those gaps.

2.1 Learning and Education

Social and individual sciences of learning have general theories and design principles (such as building explicitly on learner's prior knowledge); I build on these fields but they do not investigate curriculum or instructional design for program tracing skills specifically.

In my work I build on an idea with generally widespread support among theories of learning: effective sequences of learning experiences start from prior knowledge and build up with practice and feedback [Bransford et al., 2000]. This includes *instructional design* — the design of a lesson to teach a particular concept or part of a skill — as well as *curriculum design* — sequencing instructional designs to teach many parts of a skill to a useful threshold.

Cognitivist theories of learning frame learning as learners acquiring mental structures used to perform tasks. These theories emphasize acquiring mental representations through experience and direct instruction, as well as practice and feedback. New information is stored in working memory [Atkinson and Shiffrin, 1968, Newell et al., 1972], which is very limited to holding five to seven or even fewer pieces; thus, for learning to occur by moving from working memory to long-term memory, instructional designers should choose the right grain size for instruction, practice, and feedback [Atkinson et al., 2000]. This is critical to avoid overloading the limited capacity of working memory (the load on working memory has been called cognitive load [Sweller, 1994, Sweller, 2010, de Jong, 2010]).

In contrast, *constructivist* theories of learning emphasize that learners do not just take in knowledge as-is from instruction; instead learners actively construct their own version of knowledge based on their prior experiences and interests, and the social context and environment [Phillips, 1995, Phillips, 2000]. Constructivism foregrounds the learner's prior knowledge, as well as social and other context, instead of generic working memory limitations. This tends to correct naive errors in cognitivism like thinking of learners as empty vessels that receive knowledge, or immediately acquire refined, expert-like mental structures from instruction. Constructivism also moves from thinking about a learner's knowledge as correct or incorrect, to the viability of a learner's knowledge for particular tasks and in particular contexts, for the learner's purpose [Howe and Berv, 2000, Von Glasersfeld, 2012].

In my design work I draw on a constructivist theory of learning with a cognitivist flavor in focusing on individual mental structures: *causal inference theory*, which describes how rapid learning from single examples can occur, with the right prior knowledge [Griffiths and Tenenbaum, 2009]. My work draws on this theory when designing my reading-first curriculum and instructional design for program tracing instruction, which I implemented in PLTutor. Causal inference theory concerns learning "a causes b" causal relationships. For instance, a child may see a switch flipped then a light goes on, and they may infer the switch caused the light to go on, just from one example. The theory identifies three key enablers for this learning: *ontology* (differentiating/recognizing entities and their causal types), *constraints* on relationships (how plausible are relationships), and *functional forms* of relationships (from how effects combine and compare, to specific forms like $f = ma$) [Griffiths and Tenenbaum, 2009]. For the light example, *ontology* includes a light and a switch as inanimate/mechanical entities, *constraints* includes closeness in time (switches tend to control things quickly), and *functional form* may include knowing the switch up position activates ("turns on") whatever it controls. With these enablers, causal learning then occurs by *observing examples of the causal relationships* [Griffiths and Tenenbaum, 2009].

The sciences of learning also show learning is deeply social [National Academies of Sciences and Medicine, 2018]; although my work focuses on teaching individual learners, my thesis work draws on these theories and may also help people join communities. Learning scientists, including

the sub-field of computer-supported collaborative learning [Stahl et al., 2006], argue that many people learn skills best when living in a culture/community where the skill is used, valued, and accessible. Major learning theorists also stress this social aspect of learning (such as Papert's "go to France to learn French" [Papert, 1980]). Situated learning theories describe learning as gradually deeper participation and recognition within a community of practice [Lave and Wenger, 1991]. Thus, providing conceptual instruction in curricula that includes knowledge about the community of practice's values and why a community's tools and practices work they way they do can improve learning.

Beyond theories of learning, general learning and education research include instructional design strategies such as scaffolding. Scaffolding is giving extra information or assistance on a question a learner might otherwise find too difficult [Wood et al., 1976]. For example, a question without scaffolding: "Write down the thesis statement of this dissertation from memory." ; a scaffolded version might be "Fill in the blanks to complete this thesis statement: a new t—— of p——— l——— k——— that includes mappings from s—— to semantics and their n—— combinations can serve as the basis for more g——— tools for learning and more p—— assessments of that knowledge." As an instructional design strategy, scaffolding can allow smaller, more granular pieces of knowledge to be practiced as a part of learning, allowing the knowledge to be taught piece by piece [Wood et al., 1976].

My work draws on multiple theories, but the framing towards teaching and assessing individual learners is constructivist with a cognitivist focus on how individual learners can construct programming language knowledge from their experiences, and the viability of that knowledge [Ben-Ari, 2001].

2.2 Assessment

My thesis also builds on related work from educational assessment.

2.2.1 *Formative Use of Assessment*

Summative use of an assessment involves measuring skills at a single point in time, usually for evaluation, such as at the end of a course. Another use of a summative assessment is to compare learning tools by comparing the average test scores of their users in a between-subjects study design [Tew and Dorn, 2013].

In contrast, formative use of an assessment uses test results to give feedback or otherwise improve learning [Scriven, 1967]. Formative assessments can help improve learning when they provide actionable information for better targeted instruction, practice, and feedback [Kingston and Nash, 2011, Dunn and Mulvenon, 2009]. Decades of evidence show formative feedback has one of the best effect sizes for learning interventions, according to a review of 800 meta-analyses [Hattie, 2008, Hattie and Timperley, 2007]. While summative and formative assessments can have similar questions (called *items* in the assessment literature), formative assessments must be much more carefully designed to precisely diagnose what learners do and do not know. In educational assessment, assessments that support precise inferences about different parts of a learner's knowledge have been called diagnostic assessments [Leighton and Gierl, 2007].

2.2.2 *Validity*

Assessment validity concerns trace back historically to concerns in psychology about whether some survey or other psychological measurement technique actually measures what it purports to measure [Kane, 2013, p. 6–7] [Cronbach and Meehl, 1955], and epistemological issues around what observations provide evidence for theoretical statements (for example, disproving a theory) [Strauss and Smith, 2009]. Validity arguments for a particular use of an assessment draw on pragmatism [Legg and Hookway, 2020] and the ancient human wisdom of asking "Why? For what purpose?".

Kane's validity framework focuses on making a *validity argument* for the effectiveness of an assessment for a specific use, such as formative assessment [Kane, 2013]; in Chapter 4 of this thesis I use Kane's framework as part of designing a precise formative assessment for program tracing

skills. Kane's validity framework involves a four-part validity argument, starting from 1) *scoring*: how items are scored, to 2) *generalizing* from the scoring model to expected performance on test tasks, to 3) *extrapolating* scores from test tasks to other tasks in other contexts (including construct claims, e.g. measuring CS1 knowledge), to finally 4) *use*: using the scoring model for some decision and evaluating the broader effects of this use.[1] Kane discusses the general importance of a precise scoring model for formative assessment: "Model-based assessments can provide relatively detailed information on the attributes (e.g., skills and conceptual understandings) that each student has mastered and not mastered, and with a small grain size, this information can be quite detailed. Such specific indications of the weaknesses in a student's mastery of a topic can be used to target instruction on those soft spots." [Kane and Bejar, 2014] To gather evidence of effectiveness for formative use of an assessment, one might give the assessment to learners, use the scoring model to give a targeted lesson, then evaluate learning [Kane and Bejar, 2014].

2.2.3 Diagnostic Assessments and Concept Inventories

Concept inventory (CI) tests are assessments with validity arguments for assessing the overall understanding of a student for a particular set of concepts [Taylor et al., 2014, Adams and Wieman, 2011, Rowe and Smaill, 2007]. The Force Concept Inventory from physics education researchers started the concept inventory line of work [Sands et al., 2018, Hestenes et al., 1992]. CIs are often presented in the form of multiple-choice tests, including "distractor" answer choices that cover frequent misconceptions; for example, for an addition problem 19+2 where not carrying the one is a misconception, a corresponding distractor answer choice is 11.

---

[1] For more detail on how Kane can be applied to advance validity specifically in computing education assessment, see my paper for an extended discussion and longer exposition of Kane's framework [Nelson et al., 2019]. For example, other approaches to validity tend to overlook how a test's score will be used, instead focusing more narrowly on statistical properties of a test such as accuracy and reliability [Moss et al., 2006, Kane, 2016]. For example, in making and doing validity studies for the FCS1 test, Tew et al. describe how they created items to cover CS1 concepts, and did item-response theory analysis and think-aloud studies to identify items with variance caused by confounds (such as poor quality distractors or difficult to understand questions) [Elliott Tew, 2010]. While this traditional approach to validity helps verify that a test measures what it intends to measure, it does not necessarily verify that the measure itself is meaningful or helpful in the world - for example, how using the measure affects a variety of learning outcomes over time.

While it is a frequent perception among researchers that concept inventories can diagnose a test-taker's misconceptions, research in educational assessment has problematized that assumption. Jorion et al. shows that validity arguments need to be made separately for diagnosing test takers, and shows the different studies and analyses needed for such claims [Jorion et al., 2015]. Sands et al. includes discussion of the same issues in physics around the original Force Concept Inventory and in concept inventories in general [Sands et al., 2018]. Santiago Roman's dissertation and Denick et al. are examples of trying to interpret a CI more diagnostically [Santiago Roman, 2009, Denick et al., 2012].

In Section 5.4 this thesis analyzes the program tracing skill dependencies of questions in a concept inventory, the Basic Data Structures and Algorithm Inventory (BDSI) [Porter et al., 2019], to motivate designing more diagnostic questions.

2.3 Teaching in Computing Education Research

While educational psychology and learning science methods and theories are useful for approaching research and design, they lack theories for the specific knowledge in computing, and curriculum and instructional designs to teach them. Discipline-based education research [Peffer et al., 2016, Council, 2012], such as computing education research (CER) [Malmi et al., 2019], has the domain-specific expertise to develop those theories and make designs for learning, from curricula to interactive tools for learning and assessments. I build on work in CER, and fill these four gaps:

- Section 2.3.1: no theory of programming language knowledge that includes the mapping between token-level syntax, semantics, and execution state, nor nested combinations of that knowledge

- Section 2.3.2: no program tracing learning tool with flexible teaching granularity that mixes conceptual instruction, program visualization, and scaffolded assessment questions

- Section 2.4.1: no program tracing assessment that precisely and systematically assesses nested combinations of that knowledge

- Section 2.4.2: no advanced programming question designs made to precisely diagnose problems with advanced knowledge (such as concurrency) versus prerequisite program tracing skills (like knowing scoping rules that determine what variables may need concurrency controls added)

2.3.1 Theories of Programming Language and Program Comprehension Knowledge

In cognitivist theories of learning, it is important to define what is taught, as it is hard for learners to infer that knowledge without focused direct instruction, scaffolding, and practice. There is no prior theory of basic programming language (PL) knowledge that consistently includes the mapping between token-level syntax, semantics, and execution state, nor nested combinations of that knowledge.

Prior general conceptual theories of computing knowledge describe high level topics and concepts and some prerequisite relationships, but not at a detailed level for knowledge of a specific PL. For example, the ACM 2013 Curriculum Guide [Joint Task Force on Computing Curricula, Association for Computing Machinery (ACM) and IEEE Computer Society, 2013] and Goldman et al's list of core concepts taught in CS1 [Goldman et al., 2008, Goldman et al., 2010] describe concepts in the CS undergraduate curriculum; these sometimes include program tracing skills as a high-level category, such as "Software Development Fundamentals / Fundamental Programming Concepts". Rich et al have a more detailed conceptual knowledge map and learning trajectories but not specific to a PL [Rich et al., 2017].

Researchers since the 1970s[2] have theorized about *how* people comprehend code [Brooks, 1983, Brooks, 1999, Hoc and Nguyen-Xuan, 1990, O'Brien, 2003, Shneiderman and Mayer, 1979, Schulte et al., 2010, Teasley, 1993] without trying to define the detailed PL knowledge people need for program tracing. This research has developed cognitive theories of comprehension processes, describing perception [Hoc and Nguyen-Xuan, 1990, O'Brien, 2003], mental structures [Brooks,

[2]Before this work people in computing also pragmatically considered how to document and comprehend programs, such as with comments and with flowcharts [Knuth, 1963, Abrams, 1968], with less emphasis on making psychological theories of how people comprehend programs.

1983, Dalbey and Linn, 1985, Mayrhauser and Vans, 1995], and differences between novices and experts [Brooks, 1983, Dalbey and Linn, 1985, Mayer, 1981, Shneiderman and Mayer, 1979]. These theories facilitate questions about perceptual strategies that novices and experts use to comprehend code, what features of code experts use to comprehend code, and what kinds of knowledge experts use. This prior theory work focuses on the high-level skills, like recognizing patterns via schemata [Soloway and Ehrlich, 1984] and beacons [Wiedenbeck, 1986], or understanding which fragments of code might need to be traced to understand some code (whereas other code may be understood by recognizing general patterns and templates).

Within that work, program tracing is a particular strategy for comprehending a program's behavior. Program tracing is the *step by step execution* of code, following a control flow path using concrete values. Tracing activities can also be part of program comprehension strategies, including building strategies at higher levels of abstraction. For instance, executing a piece of code on a series of specific inputs may guide the learner to understand what the given code computes [Izu et al., 2019]. Tracing can be done mentally or with pencil and paper. Program tracing tasks can include, for example, predicting the output of a segment of code, or making diagrams representing the contents of memory after some code executes. In this thesis program tracing refers to humans using PL knowledge, although one may also use a debugger or other program visualization tool to perform program tracing tasks.

2.3.1.1 PL Knowledge

Prior theories of PL knowledge include syntax and semantics knowledge, but do not include the mapping from token-level syntax to semantics and program execution state. Prior work makes key distinctions between writing, syntax, and semantic knowledge (for example, [Shneiderman and Mayer, 1979, Hoc and Nguyen-Xuan, 1990, Bayman and Mayer, 1988]) but lacks connections across this knowledge and a principled way to derive it. Operational semantics rules and abstract machines specify how to compute in a rich formal notation [Pollock et al., 2019, Berry, 1991]; these expert notations don't specify a way to decompose a semantic rule into granular chunks for novices. For example, Berry had to add extra parts to his visualization system to avoid incomprehensible jumps

during stepping [Berry, 1991], and operational semantics also does not include parts of basic PL knowledge like syntax errors.[3] In contrast, Mayer divides semantics into micro (statement) and macro (program) levels, and describes *transactions* at a sub-statement level as action, object, and location [Mayer, 1979b, Mayer, 1981, Mayer, 1985]. However, these natural language descriptions lack connections to the sub-statement parts of the code that causes them; for example, in Table 3 in [Mayer, 1979b] for the `LET address=number` statement, the transaction that sets the address "Create Number Memory Write the new number in that memory space" lacks a connection to the `address=` .

2.3.1.2 Knowledge Models and Nested Combinations

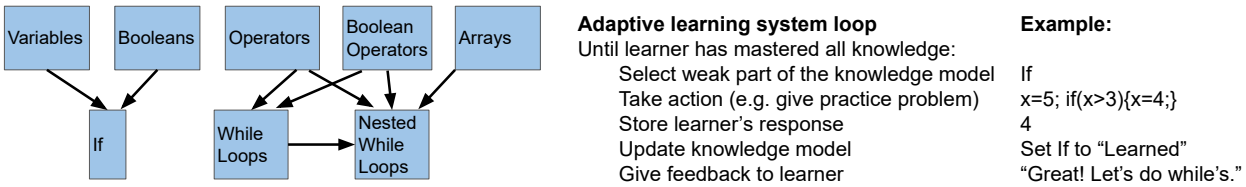


Figure 2.1: Example knowledge model and problem selection algorithm for a simple adaptive learning system. The *knowledge model* represents learner's knowledge of the domain, with the smallest unit called a *knowledge component* (KC) (such as "If"). Building on the *knowledge model*, a *curriculum model* may also includes relationships among KCs (such as pre-requisites, different curricular orderings), serving as an input to curriculum design or automated teaching systems.

A *knowledge model* models what a learner knows and acts like a mini-theory of the skill; prior knowledge models in adaptive learning systems have not attempted to systematically model the composable, nested structure of PL semantics knowledge (for example, `if` statements can be nested inside other `if` statements).

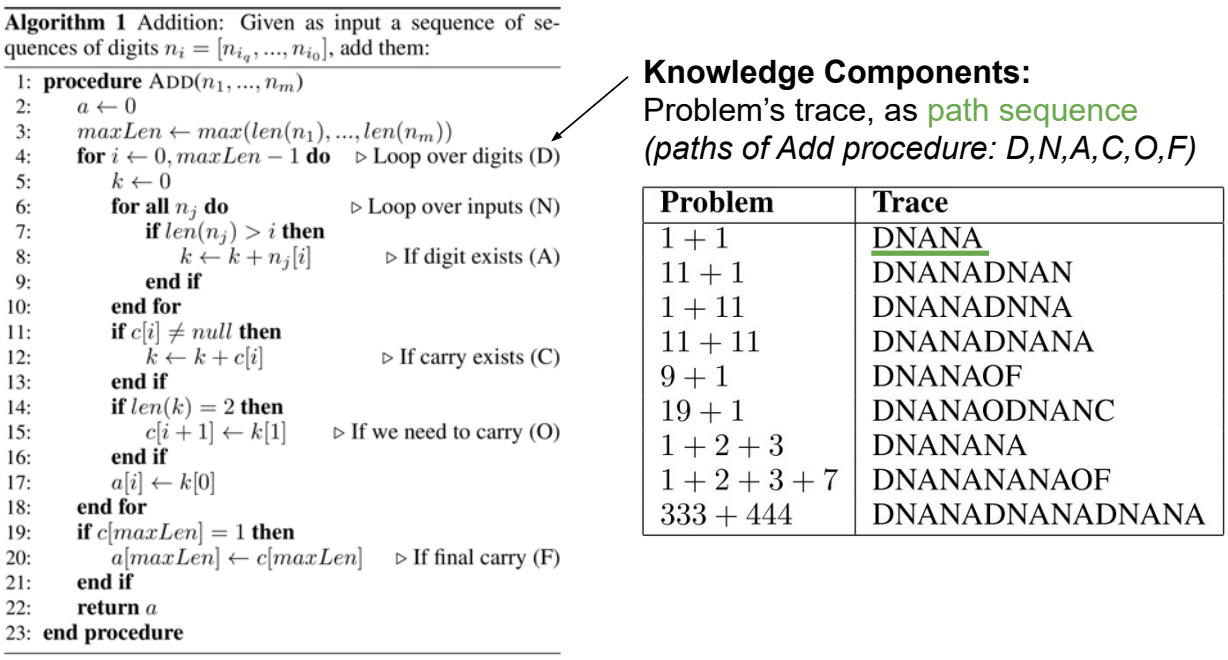
[3]The idea that novices learn general independent rules and fluently combine them can easily be a cognitivist-style error in thinking novices directly perceive and learn an expert form of basic PL knowledge. Such learning might happen for some people when learning elegant pure functional languages with simpler semantic models, but, even at institutions with carefully designed curricula to teach them, problems combining this knowledge for scoping and other combinations of rules are observed [Fisler et al., 2017].

People have built learning software with integrated curricula and assessments, starting with early computer-assisted instruction tools [Danielson and Nievergelt, 1975, Schurdak, 1967, Friend, 1973] to cognitive tutors [Farrell et al., 1984], which model the learner's knowledge to varying degrees. Figure 2.1 (left) shows how early learning systems encoded knowledge components as nodes in a graph (such as one node for "if statements" and another for "while loops"), with edges for pre-requisite relationships [Goldstein, 1979, Barr et al., 1975b] (similar to the modern day Khan Academy). On the right, Figure 2.1 shows how a learning system might use this model for learning.

While in principle a person could manually create a set of knowledge components representing all the compositions (such as nodes for "if statements inside a while loop" and other compositions), I have not found prior work that has done so systematically; instead they focus on program writing, or make the cognitivist error that learners absorb an expert form of PL knowledge as independent components easy to fluently combine. The first computer-assisted instruction systems including tracing and writing practice [Schurdak, 1965, Schurdak, 1967, Friend, 1973] but were followed by more adaptive learning systems focused on teaching writing [Barr et al., 1975b, Johnson and Soloway, 1984, Anderson et al., 1984]. Work around the 1980s with adaptive tutoring systems included teaching PL semantic rules with program tracing tasks, but it did not model learner knowledge of compositions of those rules [Anderson et al., 1989, Sjoerdsma, 1976, Brusilovsky, 1992]. Kumar has more detailed tracing skill models based on concept maps, but does not model nested combinations [Kumar, 2006]. Some recent work has critiqued the common practice in knowledge modeling to not model combinations of skills [Long et al., 2018], but has not been applied to programming and doesn't discuss systematically assessing those combinations.[4]

To generate a more detailed knowledge model from a solution procedure, Andersen created a method to create knowledge components for a domain from a solution procedure [Andersen et al., 2013], but did not represent the composable/sub-problem relation that recursive solution procedures encode (such as PL interpreters). The solution procedure takes a problem as input and outputs

[4] I don't consider myself an expert on the entire automated tutoring system literature, so I might be missing something here. There might be some work between [Shih and Alessi, 1993] and [Castro-Schez et al., 2020] I am not aware of, though I have searched for it.



Algorithm 1 Addition: Given as input a sequence of sequences of digits $n_i = [n_{i_q}, ..., n_{i_0}]$, add them:

```
1:  procedure ADD(n_1, ..., n_m)
2:      a ← 0
3:      maxLen ← max(len(n_1), ..., len(n_m))
4:      for i ← 0, maxLen − 1 do          ▷ Loop over digits (D)
5:          k ← 0
6:          for all n_j do                ▷ Loop over inputs (N)
7:              if len(n_j) > i then
8:                  k ← k + n_j[i]        ▷ If digit exists (A)
9:              end if
10:         end for
11:         if c[i] ≠ null then
12:             k ← k + c[i]              ▷ If carry exists (C)
13:         end if
14:         if len(k) = 2 then
15:             c[i + 1] ← k[1]           ▷ If we need to carry (O)
16:         end if
17:         a[i] ← k[0]
18:     end for
19:     if c[maxLen] = 1 then
20:         a[maxLen] ← c[maxLen]         ▷ If final carry (F)
21:     end if
22:     return a
23: end procedure
```

Knowledge Components:
Problem's trace, as path sequence
(paths of Add procedure: D,N,A,C,O,F)

Problem	Trace
$1 + 1$	DNANA
$11 + 1$	DNANADNAN
$1 + 11$	DNANADNNA
$11 + 11$	DNANADNANA
$9 + 1$	DNANAOF
$19 + 1$	DNANAODNANC
$1 + 2 + 3$	DNANANA
$1 + 2 + 3 + 7$	DNANANANAOF
$333 + 444$	DNANADNANADNANA

Figure 2.2: An example of how to model procedural knowledge for a non-recursive Addition procedure. Left hand side is from Andersen's work [Andersen et al., 2013].

the solution (see left Figure 2.2). The key problem is how to decompose the procedure to model the knowledge as separate parts; the solution is to model knowledge as "control flow paths" of the procedure, with each path as the smallest unit of a knowledge component (for example, C in Figure 2.2). A particular problem's knowledge component is represented as the sequence of paths encountered as executed in the solution procedure (for example, see Figure 2.2 1+1 with knowledge component DNANA). Andersen's method works for procedures describing addition and other flat procedures, but Andersen's technique has never been generalized to problem solving procedures that call themselves to solve sub-problems (i.e. are written recursively), nor does his model represent that composability explicitly and systematically. For example, flat sequences of control flow paths do not sufficiently represent the knowledge required for program tracing. For example, *Program A*: `x=2; if(x==2){x=x+1;}` and *Program B*: `x=2; if(x==2){} x=x+1;` these programs have the same sequence of control flow paths: *set-variable, if, compare variable, set-variable*. However, a person who ignores if statements would trace *Program B* right, yet that same knowledge would

fail for *Program A*. A more precise model with nested combinations might show Program A as *set-variable, if, compare variable, {set-variable}*; and Program B as *set-variable, if, compare variable, set-variable*.

In summary, there is no theory of PL knowledge that consistently includes the mapping between token-level syntax, semantics, and execution state, nor nested combinations of that knowledge. This thesis fills the mapping gap in Section 3.1 and nested combinations in Section 4.2.2.

2.3.2 Interactive tools for program visualization and learning

Prior programming tools for debugging and program visualization show how programs execute, and learning tools include different types of practice/assessment for program tracing knowledge. However, there is no program tracing learning tool with flexible teaching that mixes conceptual instruction, program visualization, and scaffolded practice questions at varying levels of granularity. From a constructivist perspective that kind of direct instruction with active learning via practice is important for enabling learning.

Prior work has built interactive learning tools with integrated curricula and assessments focused on program writing, without showing program execution visualization. This work starts with early computer-assisted instruction tools [Danielson and Nievergelt, 1975, Schurdak, 1967, Friend, 1973], and adaptive learning tools like cognitive tutors [Farrell et al., 1984, Anderson et al., 1987]. More recently, people have designed more engagement focused and accessible writing tutorials [Kalelioğlu, 2015, Codecademy, 2016].

My thesis work builds on work from prior program visualization tools which increased visualization granularity via smaller execution steps.[5] Traditional print output from a program shows one step at the whole program level; if the learner has enough PL knowledge they might infer which lines generated which output (not good for novices). Visualization tools make hidden state more explicit, showing execution at the line level [Guo, 2013b, Karavirta et al., 2015, Virtanen et al.,

[5] My thesis does not make a contribution to program visualization itself, see Baecker for a mostly post-1960's review (with some pedagogical discussion) [Baecker, 1998] and please also read Evans & Darley for before that [Evans and Darley, 1966, Baecker, 1975].

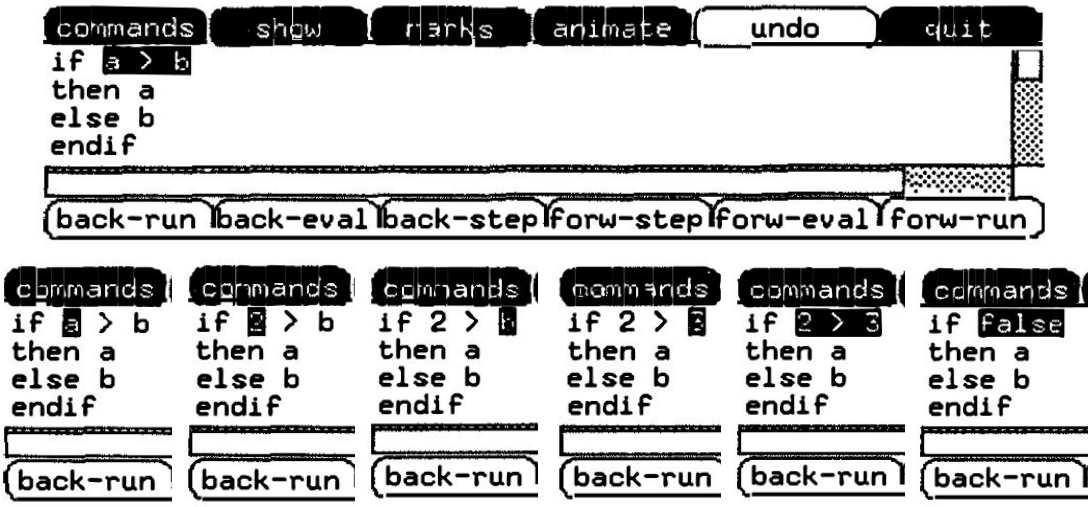


Figure 2.3: An example of sub-expression stepping granularity from Berry, with the interface at top and stepping sequence cropped below [Berry, 1991].

2005] to the sub-expression level [Baecker, 1975, Berry, 1991, Moreno et al., 2004, Clements et al., 2001, Clements, 2006, Lieberman, 1984].[6] Python Tutor [Guo, 2013b] visualizes the execution of Python programs at the line level with detailed memory diagrams; line level granularity is common in debugging IDE tools. In contrast, sub-expression level granularity shows individual expression evaluation, for example, see Figure 2.3; Berry's work here was pioneering in generating animated program visualizations from operational semantics, a formalism used by PL researchers initially for reasoning and proofs [Berry, 1991]. Sorva et al provides a further review of program visualization techniques themselves for learning, without integrated curricula [Sorva et al., 2013b], and Hidalgo-Cespedes et al critiques program visualization systems with respect to a list of constructivist learning principles [Hidalgo-Céspedes et al., 2016].

Other visualization tools for novices fit within a writing tool, such as an integrated development environment [Hendrix et al., 2007, Kölling et al., 2003, Cross II et al., 2002, Lee et al., 2013]; these show program execution but very few have conceptual instruction and none have granular program tracing assessments. The Basic Instructional Program combined program writing practice and

[6] Older videos showing program visualization in ZStep and other tools can be found on the CHI youtube channel.

visualizing the written program at the line level, without tracing assessments [Barr et al., 1975b]. Similarly, ITEM/IP shows program visualization but only assesses tracing at the whole program level, asking for the ending state of the tape of a Turing machine for a given program [Brusilovsky, 1992]. OU Learningworks has a kind of visual program simulation without assessment or tracing practice, by using a UI to send messages to objects (see figure 5 in [Woodman et al., 1999]) and provides a notional machine by filtering the Smalltalk-80 language viewers (e.g. not showing the Object class) [Woodman et al., 1999]. Csmart has a writing curriculum, giving single line writing practice problems, but interestingly shows a problem prompt as a line-level visualization showing the execution of the line to be typed, optionally with a natural language prompt [Gajraj et al., 2011].

Some program visualization and debugging systems focus on compilation and language execution, but do not integrate a curriculum or assessment. VAST shows compilers constructing syntax tree during parsing, particularly recovering from syntax errors, for the purpose of helping visualize and debug student-written parsing programs [Almeida-Martínez et al., 2009]. LPTools helps compilers students visualize from parsing to program execution, including intermediate code generation; while this technically shows mappings from token-level syntax to semantics, its level of detail would be overwhelming for novices and execution rules are represented with code [Golemanov and Golemanova, 2020]. LanugageGame is a direct manipulation program for Squeak for learners to construct their own programming language by direct manipulation of syntax and semantic rules, with some debugging and parse tree visualization features [Yamamiya, 2003].

While this prior research in program visualization for computing education has explored stepping granularities from the line to sub-expression level and what machine model to show, prior work has not designed a learning tool with a curriculum including conceptual instruction, program visualization, and practice for program tracing skills. Most visualization tools lack conceptual instruction and integrated practice, such as the Racket stepper [Clements et al., 2001, Clements, 2006], because they are designed for use within a course or while reading a textbook; this requires more motivation and self-regulated learning skills to coordinate using them together, and watching program execution visualization alone is not sufficient for learning [Naps et al., 2002, Sorva et al., 2013b]. The LISP Evaluation Modeler shows sub-expression evaluation by interactively clicking

on code to expand and evaluate, without practice or assessments [Mann et al., 1994]. LISP-KIE is a practice tool without instruction that scaffolds learners to learn code testing skills, which involves code comprehension skills, via a CodeProbe tool that provides a simplified interface for running code and getting its results, with the code of the program hidden or visible [Bell et al., 1994]. Kumar's Problets collection of tutors for mostly tracing practice lack conceptual instruction [Kumar, 2015, Kumar, 2016]. Similarly, Reduct lacks conceptual instruction, and has a curriculum with indirect assessments of program tracing knowledge, as it evaluates all program fragments for the learner [Arawjo et al., 2017]. RAPITS, an early electronic textbook for loops, included conceptual instruction and practice, but showed an example program with output instead of visualizing execution [Woods and Warren, 1995]. UUhistle has some technical features that would support a learner independently following a reading curriculum (e.g., a menu that shows a set of programs with instructor annotations), but that has not been evaluated [Sorva and Sirkia, 2010].

At a technical level, these tools lack flexible teaching granularity: a lesson designer cannot specify randomly navigating to any part of the program to show conceptual instruction or program execution or an assessment. While some systems show automated natural language descriptions for each execution step [Karavirta et al., 2015] or allow instructors to write their own [Virtanen et al., 2005], nearly all prior work shows at most one natural language explanation or assessment for each execution step, coupling them one to one — except UUhistle [Sorva, 2013b] and Kelmu [Sirkiä, 2018]. Even in those systems, these steps remain coupled during execution; UUhistle only allows extra explanation steps before the program begins executing and couples them with an early statement in the code (like a using namespace statement). Kelmu has made an annotation layer on top of a program execution steps, to add sub-steps of conceptual instruction or skipping over some number of steps, without fully decoupling from the control flow ordering of program execution; for example, a lesson cannot briefly navigate to each part of a program and describe them, then do a second pass from the beginning to go through execution with more detail [Sirkiä, 2018].[7]

Fundamentally, these tools lack flexible teaching granularity because their lesson representations

[7]More precisely, you cannot show a teaching step for execution step 1, then execution step 3, then show another for execution step 1.

have been coupled with the sequence of program execution steps, or have no coupling (like a long paragraph of conceptual instruction followed by an independent program execution widget in an interactive textbook [Sorva and Sirkiä, 2015]).

This thesis fills this gap in Section 3.2.2.1 by adding a layer of abstraction on top of program execution steps, to enable flexible teaching at the right level of granularity to learn program tracing knowledge as mappings from syntax to semantics and execution state.

2.3.2.1 Scaffolded Practice

In learning tools that show program visualization, designers cannot specify scaffolded assessments that show partial program execution or hide a subset of machine state.

Existing tools support a variety of assessments. The most basic in-tool assessment techniques are pop-up, hard-coded multiple choice questions and structured response questions (like select the elements in a table that are in an array) [Myller, 2007, Karavirta et al., 2015, Sudol-Delyser et al., 2012, Naps, 2005, Bruce-Lockhart et al., 2009]. Some practice tools have quiz questions designed to practice program tracing knowledge, like choosing inputs or parameters that will force control-flow to visit highlighted statements [Hoffman et al., 2011], but they do not show program visualization or feedback for students beyond correctness or natural language hard-coded messages. Quizjet allows authoring of parameterized tracing problems that ask for output or the final values of variables, thus limiting the granularity of questions [Hsiao et al., 2010, Hsiao et al., 2008].

UUhistle [Sorva et al., 2013a, Sorva et al., 2013c] and other systems [Kollmansberger, 2010, Kumar, 2015, Kumar, 2016] have visual program simulation, in which learners use graphical controls to execute the steps of a program using direct manipulation. The program and program execution state are filled in, but then the right part must be moved or selected then replaced with what it evaluates to (see Figure 2.4).

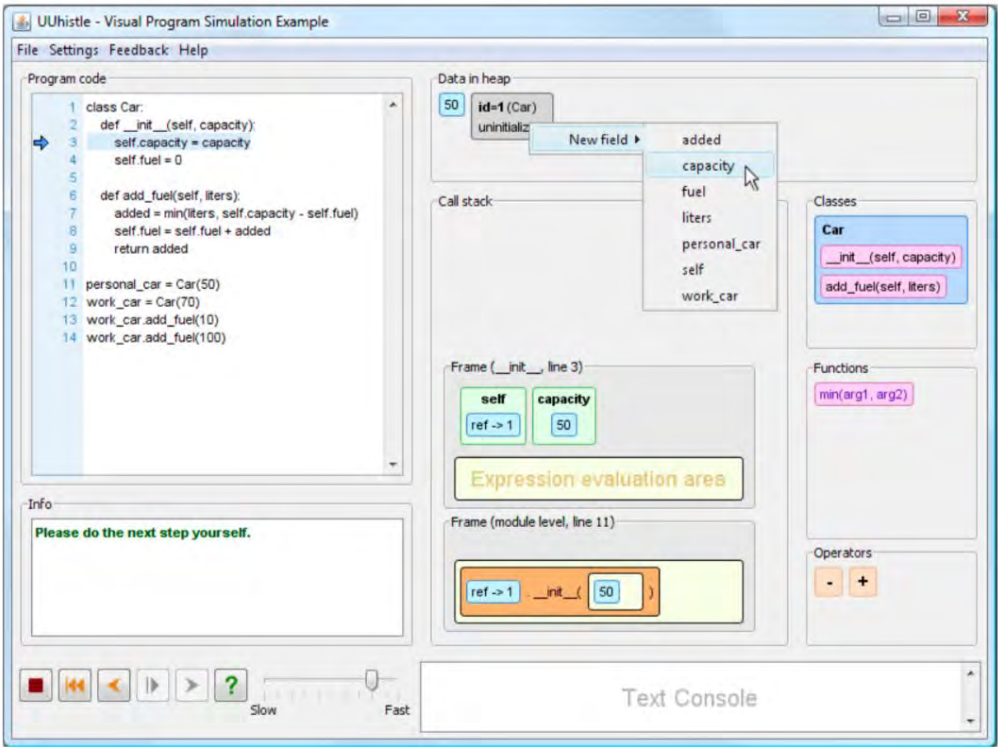


Figure 2.4: An example of a visual program simulation exercise in UUhistle. The learner is about to make the field capacity in the Car instance, then assign a value to it.

Similar but inverted, Reduct [Arawjo et al., 2017] has the learner fill in program fragments then Reduct evaluates them; this is a kind of writing puzzle, plus some clicking on expressions to evaluate them to see if the fragments reduce to the puzzle's goal state (such as reducing to two key icons which represent true values in Figure 2.5); to get to the goal state the learner needs to formulate a plan using their mental model of language semantics, or with trial and error. See Figure 2.6 for an example of a Reduct puzzle that doesn't involve writing, just evaluation.

Figure 2.5: An example of a Reduct puzzle that involves writing by choosing which blocks to drag from the bottom bar to use to fill in the code fragments on the main area and be evaluated (or compose and evaluate those blocks on their own as a separate expression).

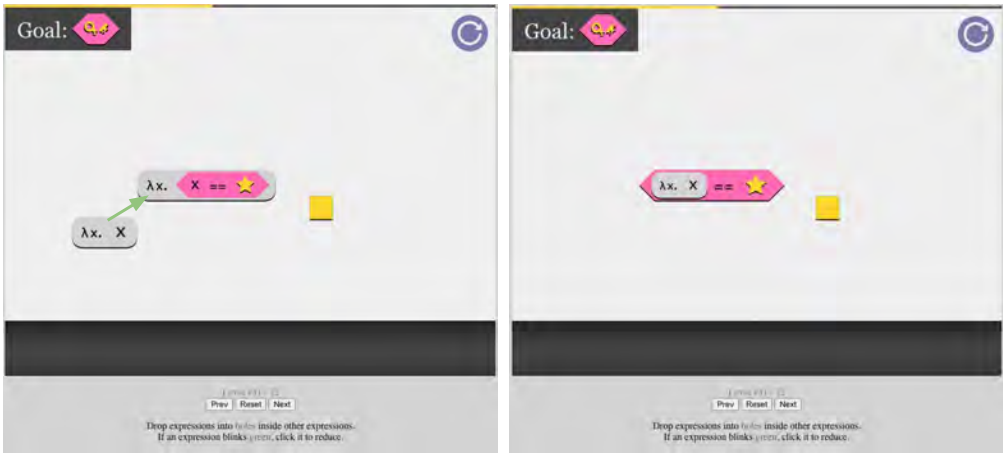


Figure 2.6: A Reduct practice exercise; the goal of the exercise is to reduce everything on the canvas to a single broken key (an analogical representation of false). One step is shown, a reduction by dragging and dropping the lambda onto the larger lambda expression (green arrow). Although Reduct does not propose this idea, here's an example of how to vary Reduct's assessment granularity: an assessment at higher-level granularity could show a circle around some fragments then ask what they reduce to; that higher granularity would force more mental execution by the learner.

While those existing techniques assess line [Myller, 2007, Karavirta et al., 2015] or sub-expression [Kollmansberger, 2010, Sorva and Sirkia, 2010, Kumar, 2015, Kumar, 2016] level PL knowledge, no learning tool can flexibly change that granularity to vary from assessing a sub-expression level step to larger sections of the program. UUhistle can do assessments for a few targeted execution steps instead of all steps in a given program, but, like all other tools, UUhistle halts execution to hide program state related to the answer, which couples assessment granularity to their execution step granularity (for example, in Figure 2.4, if the learner could just advance execution one step, they would see the answer). CWadeIn and WadeInII adapts which levels of expression evaluation are shown, stepping over lower-level steps already learned but without hiding their results; both also only works for expressions [Brusilovsky, 1992, Brusilovsky and Su, 2002, Brusilovsky and Loboda, 2006, Loboda and Brusilovsky, 2008, Loboda and Brusilovsky, 2010].[8] Published in 2018 after my work, Bauer et al's Practicum tool provides different levels of

[8]Some other program visualization tools allow the user or use automation to adapt the level of detail, but lack

scaffolding but does not show partial program execution while hiding state [Bauer et al., 2018].

In summary, existing learning tools with curriculum do not support flexible teaching granularity, which requires decoupling teaching a program from the execution of that program. They do not include instructional design that mixes conceptual instruction with program execution and assessments (which this thesis fills in Section 3.2.2.1), or scaffolded assessments that hide some program state while showing other parts of program execution (which this thesis fills in Section 3.2.2.2).

2.3.3 Curriculum and Instructional Design for Program Tracing

Given that there is no theory of PL knowledge as knowing mappings from token-level syntax to semantics and execution state, there are no curricula systematically designed for teaching that knowledge via human causal inference. The gap in prior work is actually a little larger: there is no reading-first curriculum design that mixes conceptual instruction, program execution visualization, and assessments. This thesis fills that gap in Section 3.2. The closest prior work has contributed the idea of starting learning with conceptual instruction on PL semantics (but without sustained and direct tracing practice), and several instructional design approaches, such as notional machines for PL semantics, and visual analogies and concreteness fading for teaching syntax and semantics.

Early[9] curricula for learning programming focuses on writing after some relatively brief conceptual explanation on the computer and the PL, although some work has described including some initial reading or tracing practice, or interleaving tracing and writing practice. The typical spiral writing approach teaches syntax and semantics one language construct at a time, with writing tasks for practice [Shneiderman, 1977, Van Merrienboer and Krammer, 1987, Robins et al., 2003]. Other instructional designs included some reading or tracing practice, but quickly included writing - for example, in the mid 1970s the learning progression for Smalltalk at Xerox PARC [Goldberg and

---

instruction or practice. For example, Fiz automates recognizing patterns in code to then show them to users, designed to be useful for novices and experts to comprehend code, but lacks any instruction or assessment features [Eisenstadt et al., 1993]. WINHipe allows specifying granularity of a program execution visualization, from the expression to higher more algorithmic levels [Pareja-Flores et al., 2007].

[9]For a brief history of the context of this work that also goes back to the 1950's, see the beginning chapters of Leonel Morgado's dissertation [Morgado, 2005].

Kay, 1976], and for ELOGO [du Boulay and O'Shea, 1976]. In contrast, a curriculum design that interleaved tracing and writing practice was used in a self-contained computer-assisted instruction system for introductory programming at Stanford starting in 1967 [Suppes et al., 1977]. The curriculum included hundreds of program tracing problems; the overall curriculum design was, for each language construct, start with a natural language explanation and code example, then tracing practice, then small scaffolded writing tasks (like filling in missing fragments) then writing problems similar to current introductory courses (short problem prompts where students must write a solution from scratch) [Friend, 1973].[10] Mayer, Dyck, and Bayman did experiments on different instructional designs for programming but not to design a reading-first curriculum; those designs interleaved writing practice and either program output questions [Dyck and Mayer, 1989] or conceptual recall about line-level program statements i.e. "to interpret a single statement or command, such as: Suppose `10 READ B` is one of the lines of a program. Which one of the following statements provides value(s) for the variable B? Circle the relevant one: (a) `50 INPUT B`, (b) `30 GOTO 20`, (c) `20 PRINT B`, (d) `40 DATA 65, 17`, (e) `60 IF B > C THEN PRINT C`" [Bayman and Mayer, 1988].[11]

---

[10]However, the follow-up project, the Basic Instructional Program (), focused on adaptively sequencing practice and giving feedback during program writing, and included line-level visualization of the program. Perhaps this was the rationale for removing all tracing practice, although the reports do not give any justification or comparison with the prior curriculum [Barr et al., 1975b]. BIP's knowledge model of the learner also didn't distinguish between writing and reading/tracing skills. This historical episode might be an interesting example in the history of computing for placing the research goal of automating the sequence of instruction ahead of evaluating the effects of that automation on the ultimate goal of the system - here, teaching programming. See [Suppes et al., 1977] for a review of work from that thread of computer-assisted instruction research.

[11]There are many descendants in this line of curriculum design, which focus on writing but give some exposure to program reading or visualizations, while often not specifying instructional designs for those skills and how direct practice and feedback are done. For example, Kolling et al describe a curriculum sequence using BlueJ, an IDE with program execution visualization, to run large meaningful programs to learn basic PL knowledge about objects, classes, and methods (without specifying how practice is done and feedback given) then quickly move to modifying increasingly large parts of code [Kölling and Rosenberg, 2001]. For an incomplete review of this work (it is quite large and much of it is in parts of papers primarily about a tool or other instructional design), see the Cambridge Handbook of Computing Education Research [Fincher and Robins, 2019] and [Griffin, 2018]. One line of work not discussed there descends from the community of practice around design patterns and pattern languages [Alexander et al.,], and is generally outside the CER community. Their collection of curriculum and lesson design patterns grew out of the design patterns for software community and conferences [Bergin,]. Their work generally does not engage with prior work in computing education and learning, although the ideas are useful and a good example of level of detail desired by a community of practice [Bergin, 2000].

Early theoretical work recognizes the importance of a learner's mental model of how the PL works and several instructional design approaches, without proposing sustained and direct tracing practice. Du Boulay articulated the curriculum design insight that the model of the computer for novices should not be the hardware of the computer but rather the conceptual machine implied by the language, calling that model the *notional machine* [du Boulay et al., 1981, Fincher et al., 2020, Guzdial et al., 2019]. For effective learning the notional machine should be simple and visible [du Boulay et al., 1981]; learning happens via conceptual explanations of the PL based on analogies and novices seeing the actions of programs they write, e.g. using line level debuggers such as in the Basic Instructional Program [Barr et al., 1975b, Barr et al., 1975a]. Wallace Feurzeig, Seymour Papert, Cynthia Solomon, and others created the LOGO robotic turtle programming environment, which made many program actions visible and provided a notional machine where the learner can imagine being the turtle, connecting to their prior experience [Solomon et al., 2020, Feurzeig et al., 1970]; LOGO was also arguably a computer culture to learn within, focused on writing and debugging, with intensive learning support from human teachers [Solomon, 1976]. Mayer recommended conceptual instruction that breaks down PL statements into smaller sub-steps called transactions, without proposing a curriculum with practice at that more granular level [Mayer, 1979a]. Mayer created an instructional design with line-level visual program simulation with a concrete physical computer model and a manual about the PL that encouraged learners to "role play" the computer [Mayer, 1976]. For Mayer this conceptual instruction and model helped organize learners' knowledge before transferring it to writing tasks [Mayer, 1979c].

In contrast to writing-focused approaches, around 1980 Deimel and Moffat [Deimel and Moffat, 1982] and Kimura [Kimura, 1979] briefly proposed a reading-first curriculum starting with running programs, looking at input and output (I/O) to infer semantics, and program tracing practice, but without granular direct visualization and program tracing assessment. This work was similar on the surface to Bork's "ten finger" curriculum of typing line-level expressions into an interpreter from a manual then learning by watching interpreter output [Bork, 1971], but instead used larger more meaningful programs, valued tracing and comprehension skills, and included program tracing

practice.[12] Those pedagogies lacked a theory of the knowledge learned, making one unable to determine when a PL has been fully covered. They also lacked assessment methods beyond I/O prediction, making it hard to give targeted practice, diagnose misconceptions, and correct them with feedback. They also lack a published implementation, though Deimel has later workbooks for higher level program comprehension practice and assessment [Deimel and Naveda, 1990].[13]

Nearly all published partial implementations of a reading-first curriculum gave conceptual instruction and practice without showing program visualization or teaching nested combinations. Instead of showing program visualization, Dyck and Mayer focused on concreteness fading, by having syntax fade from natural language pseudocode to BASIC code. For practice they used only two kinds of multiple-choice computerized questions: choosing which of n programs produces a given output, and which of n outputs a given program produces. They found positive transfer effects to program writing skills [Dyck and Mayer, 1989]. Shih and Alessi built on this work and compared with a writing-focused curriculum, finding the reading group better on tracing questions and writing group better on writing questions, but less positive transfer from the reading group to writing [Shih and Alessi, 1993]. A later spiral writing curriculum includes some tracing practice for each construct and has initial evidence for effectiveness in teaching writing skills and reducing failures in CS1 [Hertz and Jump, 2013].

---

[12]While this thesis makes contributions to the initial part of the reading stage, for context Deimel and Moffat's curriculum starts with learners running large meaningful programs as users (using the programs), then reading them, then modifying them, then finally to full writing skills by creating programs. This seems very similar to the Use-Modify-Create framework for instruction from [Lee et al., 2011], a major computational thinking paper a few decades later with about 600 citations. Deimel and Moffat's earlier idea has an extra stage before Modify, more appreciation of tracing and comprehension skills, and direct practice for those skills. Recent work has a guided worksheet approach that perhaps includes some of the reading stage, called TIPP and SEE, which prompts learners to make targeted changes to the program and run the program to infer what is happening (a constructionist approach) [Salac et al., 2020]. The reading part is guiding the learners through a program behavior localization strategy by clicking on sprites in Scratch which then show code that involves that sprite.

[13]Unfortunately major literature reviews have somehow failed to mention these early works by Deimel and Moffat, and Kimura, even since the 1990s [Pears et al., 2007, Robins et al., 2003, Winslow, 1996]. The "immersion" approach also rediscovers this same early large program approach, but adds students writing natural language reflections about reading and writing code [Campbell and Bolker, 2002]. Some other later interesting works by Deimel include teaching program comprehension to professional development for software developers [Rifkin and Deimel, 1994, Rifkin and Deimel, 2000].

The closest work to implementing a reading-first curriculum is Reduct, which shows program execution and has a curriculum, but Reduct does not include direct tracing practice and only teaches the individual semantic operations of the language (with no coverage of traditional sequential code arranged as a program, and no programs with meaningful social purpose) [Arawjo et al., 2017]. Reduct's curriculum does not cover variables, programs executing according to statement order (i.e. one line after another), scope, or nested statements (lambda compositions are immediately evaluated in Reduct, and nested lambdas with unbound parts are disallowed by the editor). Reduct's design focuses on concreteness fading as an instructional design, targeting analogistic transfer to programming notation.[14] Reduct has some direct instruction by introducing new blocks via an easy puzzle where the learner composes blocks to see execution happen, similar to stepping through an example. At the same time, the curriculum design lacks natural language conceptual instruction and doesn't draw on social theories of learning; there is no discussion of the design of the PL, why it works that way, or showing example programs with meaningful uses.[15]

Lastly, some work implements a spiral writing curriculum that includes tracing practice at the whole program level of granularity; this curriculum uses writing practice primarily and lacks tracing practice for larger meaningful programs (instead it shows program execution visualization during advanced writing practice). ITEM/IP is an adaptive teaching tool for a Turing machine automata language (Turingal), with Pascal-like syntax and a Turing machine as notional machine [Brusilovsky, 1992]. ITEM/IP is similar to the Basic Instructional Program [Barr et al., 1975a], but with a spiral writing curriculum with tracing practice. ITEM/IP starts teaching PL constructs with conceptual instruction, then a separate visualization of whole program examples without natural language explanations for each step, then tracing practice at the whole program level (effectively asking what the output is), then writing tasks. Writing tasks have unit tests as input-output pairs, with

[14]Their design uses physical metaphors, which might be essential as ontology/causal type information, according to my theoretical description of learning based on causal inference in Section 3.2. To see the specific metaphors, at the time of writing one could play the original Reduct and a more advanced version without the metaphors which ends with an interface similar to TileScript [Warth et al., 2008].

[15]Reduct's approach is most similar to Bork's "ten finger" approach from 1971 [Bork, 1971], but with more granularity, and more engaging practice with concreteness fading.

help given as "rough plan for solving problems" and hints as program visualization of solution executing without showing solution code. Writing correctness feedback is "explained" by showing visualization of the student's program on the test case. Lastly, there is an analysis task giving a problem and a model solution which then has test case runs visualized, then the student can experiment with it; from the text of the paper it is unclear if these are partial or fully working solutions, but they are definitely not direct assessments of tracing or program comprehension with feedback.[16]

Intellectual descendants of this system use these curriculum sequencing ideas and whole program input/output style of tracing questions. For example, Item/IP-II adds adaptive visualizations given what the system thinks the student knows, without providing tracing practice at those varied levels of granularity [Brusilovsky, 1994]. An adaptive version of ELM-ART features a component that visualizes expression evaluation at the sub-expression level, without corresponding assessments at those levels [Weber and Brusilovsky, 2001, Weber and Brusilovsky, 2016]. [17]

My thesis work was also inspired by prior findings and arguments about the pedagogical importance of understanding PL knowledge when learning programming, which I describe in the next two paragraphs.

Recent empirical work shows writing-focused pedagogy may not teach program comprehension and program tracing skills well. Prior computer science education research has focused on writing, and assumed that comprehension skills were well learned, perhaps based on early work on program comprehension and bugs which attributed these errors to poor writing knowledge in the 1980s and individual differences in learning outcomes (some people do learn) [Pears et al., 2007, Robins et al., 2003, Winslow, 1996]. However, recent multi-national assessments indicate poor tracing

[16]For tracing practice, ITEM/IP specifies each teaching action at the whole program level of granularity, rather than a *part of the execution* of a program.

[17]These systems record how often learners have seen visualizations at various levels of granularity [Brusilovsky, 1994], automatically making them higher level over time without directly assessing gradual increased fluency with tracing skills. The later Elm-Art and Elm-Art II focus more on adaptive hypermedia navigation for the learner, diagnosing writing problems, adaptive conceptual instruction content, and interfaces for the learner models so learners can correct and customize what they see. The initial implementation of the learner model did not require radical changes, and it is coupled to the curriculum design, storing engagement and completion for exercises and content at each stage.

knowledge [Lister et al., 2004b, McCracken et al., 2001]. Other studies correlate poor reading skills with poor writing performance in introductory courses, suggesting tracing knowledge is critical for learning writing [Lister et al., 2009, Venables et al., 2009, Lopez et al., 2008, Izu et al., 2019, Clear et al., 2011]. The relationship between writing, tracing, and reading skills in CS2 have been investigated in [Corney et al., 2014, Harrington and Cheng, 2018, Pelchen et al., 2020] with conflicting results, which may indicate CS1 students with poor reading skills advance less often to CS2. There are even reports in the literature of explicit rescue interventions at the end of a course that provide substantial tracing practice, such as an award winning paper at ITiCSE in 2019 [Cutts et al., 2019].

More recent theoretical work by Sorva and others [Sorva, 2010, Zander et al., 2008, Shinners-Kennedy, 2008, Vagianou, 2006] argue that program dynamics (how static code causes dynamic computer behavior) is a threshold concept, essential for accessing other programming concepts. Sorva discusses theoretical [Ben-Ari, 2001] and further empirical [Mayer, 1976, Mayer, 1981] arguments for "...help[ing] students form a viable mental model of the computer before they are taught to program in a high-level language...to prevent the rise of misconceptions...". They describe this as a pedagogical goal, and give high-level principles without proposing a detailed curriculum.

2.4 Assessment in Computing Education Research

2.4.1 Precise assessment

Published research in computing education describes a variety of assessments, mostly on skills in the first two introductory college courses (CS1 and CS2) but also some assessments for advanced skills such as data structures and algorithms. These assessments lack precision to diagnose a learner's PL knowledge, as they do not systematically test nested combinations of PL knowledge (for example, nested if statements). This thesis fills that gap in Chapter 4.

Prior work has assessed CS1 skills, which often include program tracing questions, but without a precise scoring model that includes nested combinations of that knowledge. Existing assessments [Syang and Dale, 1993, Tew and Guzdial, 2011, Parker et al., 2016, Caceffo et al., 2016, Snow et al.,

2017, Zur-Bargury et al., 2013, Decker, 2007, McCracken et al., 2001, Lister et al., 2004a] lack a *precise scoring model* because their single numeric score output is difficult or impossible to map to specific feedback (e.g., when a learner gets 7 of 27 questions correct on the SCS1 [Parker et al., 2016], it is unclear what they need to learn to improve). Several works have studied item difficulty and item measurement characteristics but for program writing or for their fitness for aggregating into a single numeric score [Luxton-Reilly and Petersen, 2017, Simon et al., 2016, Sheard et al., 2016, Zingaro et al., 2012, Xie et al., 2019a]. For a general review of works on assessment for introductory programming and program writing, the reader may refer to [Luxton-Reilly et al., 2018b, Lancaster et al., 2019].

Concept inventories for computing topics try to identify known misconceptions [Taylor et al., 2014, Caceffo et al., 2018, Porter et al., 2019]; while CIs might have an implicit scoring model of the learner's misconceptions via which distractor answer the learner chose, they lack evidence for such use and principles for systematically and precisely assessing potential misconceptions. In principle one might use a program tracing misconception catalogue [Sorva, 2012] as a part of making a concept inventory for program tracing skills but no work to my knowledge has done so.

Ultimately, these assessments lack precision because the granularity of each question is too large: one question can require many pieces of knowledge, which inhibits inferring why the learner got the question wrong to give precise feedback. Many course exams and even published research assessments of programming knowledge have items that require 10-20 concepts to answer correctly, even for basic skills like program tracing [Luxton-Reilly and Petersen, 2017, Petersen et al., 2011, Tew and Guzdial, 2010]. The 2017 ITiCSE working group on assessing programming fundamentals also showed how a single question can assess many skills/parts of knowledge [Luxton-Reilly et al., 2018a]; for example, missing one piece of knowledge, like not understanding modulus operators, can cause learners to get large questions incorrect [Kennedy and Kraemer, 2018].

Some work has proposed slightly more precise scoring models by having questions for different high level categories of knowledge, such as writing, program comprehension, and syntax. Deimel mapped program comprehension questions to covering different levels of Bloom's Taxonomy, without specifically covering program tracing [Deimel and Makoid, 1985, Deimel and Naveda,

1990]; more recently work puts program tracing questions as a category into a level of Bloom's Taxonomy [Gluga et al., 2012]. Lemos discusses testing "grammatical" knowledge (syntax and semantics) and program comprehension at the level of general types of questions that might be used, like multiple choice and free response questions [Lemos, 1980]. Lemos also discusses using a single large program to test debugging skills without directly scoring program tracing skills [Lemos, 1980]. Similarly, Koltun describes ideas for testing program comprehension at the word, sentence, and essay levels, generating ideas for assessment by analogy from reading tests, and a long sample exercise that spans high level purpose down to tracing questions at the input/output level [Koltun et al., 1983]. The Canterbury question bank and others collect items and try to classify items to a conceptual scoring model which has some broader program tracing categories [Sanders et al., 2013, Giordano et al., 2015, Gluga et al., 2012]. A 2019 ITiCSE working group catalogued many kinds of program comprehension tasks, including program tracing tasks, but not more precise scoring models for those questions [Izu et al., 2019]. Zingaro et al propose more precise writing "concept questions" which test knowledge for small templates like "iterating over a list" [Zingaro et al., 2012], but not tracing questions.

Within assessment work on program tracing, some work designed questions to measure the general developmental stage of a learner's PL knowledge; this is at a higher level compared to diagnosing program tracing knowledge precisely. First, the BRACElet project investigated "Explain this code in plain English" questions, asking for natural language summaries of behavior at a conceptual level (e.g. averaging numbers) and used Bloom and SOLO taxonomies to analyse them [Whalley et al., 2006, Clear et al., 2011]. Second, a small set of work tries to identify the neo-Piagetian developmental stage of a learner [Lister, 2016, Teague and Lister, 2015, Teague et al., 2015]. While identifying growing mastery via presence of abstract tracing skills is promising as a meaningful assessment of development in tracing knowledge, it is a different kind of knowledge theory, not precise for the given PL and its parts and nested combinations.

For program tracing knowledge the most precise question designs cover different syntax combinations, but not systematically generating questions to cover nested combinations of syntax and semantics. The most precise scoring model implemented by an assessment is in a self-reported

Likert rating of program comprehension knowledge at the level of functions, objects, loops, variables, I/O, data collections, and conditional statements [Duran et al., 2019]. The most precise question designs are in [Luxton-Reilly et al., 2018a], derived to test parts of the grammar of the PL; this includes a question for nesting via syntax (`if` inside an `if`) but not considering different semantic cases (i.e. the outer `if` condition is true and the inner `if` condition is true versus the outer condition is false and inner condition is true - a novice learner might know one but not the other)[18] nor systematically generating questions to cover nested combinations.

Some work generates program tracing questions but does not to combine them to make a precise systematic assessment for a PL. COLT generates tracing problems so that the final answer depends on all the variables, which have manipulations generated for them, but otherwise these problems are completely random and their learner model lacks nested combinations [Myers and Chatlani, 2017]. Zavala and Mendoza generate more engaging questions that use semantically meaningful data like selecting a city from a list of actual city names, using semantic ontologies; their item templates are not systematically designed to cover nested combinations [Zavala and Mendoza, 2018]. Thomas et al. generate program tracing questions that do include smaller nested combinations up to large compound questions assessing 10-20 parts of PL knowledge, without systematically selecting those questions to cover the PL with questions small enough to be precise [Thomas et al., 2019]. Thomas also does not include function or scoping questions.

2.4.2 Advanced skills and program tracing skills

In computing education, work focuses on the assessment of either basic skills or advanced skills (classes following the CS1 CS2 introductory sequence, such as Data Structures and Algorithms). In contrast, my *differentiated* assessment questions are designed to do both, increasing precision. There are no published advanced programming question designs made to precisely diagnose problems with advanced knowledge (such as concurrency) versus prerequisite program tracing skills (like knowing scoping rules that determine what variables may need concurrency controls added). This

[18] Perhaps because prevailing theories of PL knowledge do not consider the unit of knowledge as mappings from syntax to semantics.

thesis fills that gap in Section 5.5.

Prior work has found weakness in advanced skills correlates strongly with poor prerequisite skills, such as program tracing skills, but used prerequisite assessment questions instead of creating new question designs. Valstar et al. showed that more than 30% of students could not do questions on pointers or tracing recursion at the start of an upper level data structures class, which correlated with final exam scores [Valstar et al., 2019]. Fisler et al. showed more than 30% of 3rd and 4th year CS majors failed questions on scope, parameter mutation, and/or variable mutation, suggesting knowledge transfer requires explicit instruction and/or reinforcement of prerequisites [Fisler et al., 2017].

Existing advanced skill assessments focus on the advanced skill instead of also diagnosing prerequisite program tracing knowledge issues. Question banks contain instructor-submitted questions that cover advanced topics [Sanders et al., 2013, Giordano et al., 2015]. Assessments have been developed for algorithms and data structures [Paul and Vahrenhold, 2013, Karpierz and Wolfman, 2014]. Assessments with explicit validity arguments have been developed for recursion [Hamouda et al., 2017]. The Basic Data Structures Inventory (BDSI) has the most extensive empirical work, covers some basic data structures and algorithm runtime analysis concepts, and was developed as a concept inventory [Porter et al., 2019].

Interactive learning tools include separate problems each for prerequisite program tracing skills and for advanced skills, instead of having joint questions designed to diagnose among the two. The closest work is the Open Data Structures and Algorithms (OpenDSA) project [Fouh et al., 2014, Shaffer et al., 2011], a complete interactive text book for DSA that includes a number of proficiency exercises that ask the student to simulate an algorithm step by step. This kind of visual algorithm simulation [Korhonen, 2010] resembles the process of tracing a fragment of code, but it is done at a higher algorithmic level of abstraction.

With the comparatively small work in computing education research on advanced skill assessments, I am only aware of one work that has analyzed assessments of advanced skills; it did not analyze if existing questions could diagnose prerequisite program tracing skills and advanced skill issues jointly. Simon et al. catalogued the types of questions on a sample of instructor-created exams

that included data structures knowledge, finding some tracing questions but a focus on implementing data structures and writing [Simon et al., 2010].

Prior qualitative work has shown prerequisite program tracing knowledge impacts student performance in particular advanced topics; this work suggests students may benefit from diagnosing program tracing prerequisites versus more advanced concurrency skills. Strömbäck et al. [Strömbäck et al., 2019] analyzed a large number of student solutions to a concurrency problem involving synchronizing a simple list-like data structure. Some student mistakes were likely due to not understanding how the synchronization primitives work, mainly semaphores and locks. Other mistakes were likely due to lacking prerequisite skills, such as lacking tracing skills and a lacking understanding of pointers and scoping of variables.

2.4.3 Formative Assessment

Computing education research has strong initial work creating assessments with validity evidence for summative measurement of a construct, such as self-efficacy [Danielsiek et al., 2017] or CS1 knowledge [Tew and Guzdial, 2010], but the opportunity to design or evaluate assessments for validity specifically for formative use remains unexplored. Decker and McGill catalogued instruments used or published between 2012-2016 by searching online and reviewing the table of contents for ICER and the CSE and TOCE journals. They found 13 instruments for knowledge of computing or computer science, 3 for CS1, 1 for CS2, 6 for computational thinking, and 3 for advanced skills; and 31 non-knowledge instruments that "...measured constructs such as self-efficacy, anxiety, confidence, enjoyment, sense of belonging, intent to persist, and perceptions" [Decker and McGill, 2019]. Margulieux et al. reviewed measurement practices in CER papers reporting qualitative and quantitative human-subjects studies from 2013-2017 and found 16 standardized computing-specific measurements used. None of these present arguments for validity for formative use of these measures, nor the computing education assessments in the previous parts of Section 2.4. I fill this gap in Section 4.2.5 and Section 4.3.

While some adaptive learning [Ericson et al., 2018] and tutoring systems [Crow et al., 2018] give feedback based on performance and may improve learning, in computing none have tried to

make validity arguments for how well the models of knowledge internal to these systems measure knowledge or skills. They do not discuss measurement validity and limitations of their model or item designs. Shute et al. shows what an adaptive formative assessment might look like in a non-computing domain [Shute et al., 2008].

Chapter 3

A THEORY-BASED INTERACTIVE TEXTBOOK FOR PROGRAMMING LANGUAGE TRACING

In this chapter I develop a theory of basic PL knowledge as knowing mappings from token-level syntax to semantics and execution state, as encoded in a PL interpreter's execution paths. I then use that theory as the basis for a new reading-first spiral curriculum approach and implement that curriculum as a self-contained interactive textbook. This work[1] was peer-reviewed and published in the ACM International Computing Education Research conference [Nelson et al., 2017].

3.1 Theory of Basic PL Knowledge

My first observation is that inside the interpreter that executes programs for a particular programming language (PL) is the exact knowledge required to execute a program. It just happens to be represented in a notation designed for *computers* to understand rather than people. For example, in many PL courses, students write an interpreter for a calculator language; it reads text such as `2+3` and executes that code. The interpreter contains definitions of execution rules like `if (operator == "+") { result = left + right; }`. I argue that this logic is the knowledge required to accurately trace program execution.

Unfortunately, because this logic is represented as code, it is not easily learned by novices. First, few materials for learning a language actually show the interpreter's logic explicitly. Moreover, even if this logic was visible, novices would not likely understand it because they do not understand the

[1] I led this work but also had wonderful collaborators. Amy advised the work and helped implement parts of PLTutor, although not the novel parts described for this thesis. Benji Xie was an enthusiastic collaborator, and led some study logistical tasks like writing the script for instructors during the evaluation study, videos, coordinating with the SCS1 team (Miranda Parker) to use their assessment, and discussions for future research ideas on program tracing and assessment. Alexandra Rowell and William Menten-Weil helped as undergraduate software developers fixing bugs and styling issues before the study. Leanne Hwa, Alex Hui Tan, and William W. Kwok later helped as undergraduate software developers and designers adding usability improvements to PLTutor.

notation that it is written in. This provides a key theoretical explanation for why learning to trace programs is hard—this notational barrier can only be overcome once you understand programming languages, creating a cyclic learning dependency.

Execution rule logic, however, is not alone a suitable account of the knowledge required for tracing. My second claim is that to know a programming language, learners also must be able to *map* these execution rules to the syntax and state that determines what rules are followed and in which situations. Therefore, knowledge of a PL is also the *mapping* between syntax, semantics, and state.

To illustrate this mapping, Table 3.1 below shows an interpreter in pseudocode, showing the three conventional stages of transforming program source code into executable instructions, for a simple JavaScript-like expression `x == 0`. The first stage translates characters into tokens; the second stage translates tokens into an abstract syntax tree (AST); the final stage translates the AST into machine instructions that ultimately determine program behavior. I argue that learners do not need to understand these stages themselves, but rather that they need to understand *each path* through these stages that map syntax and state to behavior. I show one example of a path underlined in Table 3.1, which specifically concerns the `0` in the `x == 0` expression, showing its translation from character to token to a machine instruction that pushes `0` onto an accumulator stack for comparison to `x` by the == operator. This simple mapping rule—that a numerical literal like `0` is a token in an expression that results in a number being pushed onto a stack for later comparison—is just one of the execution rule paths; I argue learners must understand all possible paths to know the whole language.

Some rules have one path (for example, the `0` in `x==0` only has one in my example language), but some execution rules have multiple control flow paths, depending on the code or runtime state involved. For example, `if` statements in many imperative languages can optionally include an `else` statement. If an AST has an `else` statement, the mapping is different, because the end of the `if` block must include a jump past the `else` block. Learners must understand these syntax-dependent branches in compilation and execution.

I therefore view the set of *all* possible paths through all of the compilation and execution rules

Stage	Transformation Rule (Pseudocode)	Example Output (input for next row)
1. Tokenize	Variable name => Name Operator => Op Any number => Number	Name(x) Op(==) Number(0)
2. Parse	Parse($toks$) => AST(Parse($toks_L$), Op, Parse($toks_R$)) ELSE AST(Name) ELSE AST(Number)	AST(AST(Name(x)) Op(==) AST(Num(0)))
3. To Machine Code	Code(AST) => IF AST_1 Op(==)AST_2: Code(AST_1) Code(AST_2) DO_EQUALS_OP ELSE IF Name: LOOK_UP_AND_PUSH name ELSE IF Number(n): PUSH n	LOOKUP_AND_PUSH “x" PUSH 0 DO_EQUALS_OP

Table 3.1: An interpreter with pseudocode notation. For input `x==0`, the example column shows instances of tokens in the 1st row, ASTs on the 2nd, and instructions on the last. An example semantic path is shown by dotted underlining for for a PUSH instruction for the number token.

in all of a programming language's constructs as the knowledge required to "know" a PL. This symbolic representation preserves the fidelity of the knowledge because I derive the knowledge directly from the interpreter. In my instructional design that follows, I then make these abstract paths concrete by presenting programs that cover these paths.

3.2 Learning Design and PLTutor

In this section I propose a reading-first curriculum and instructional design based on my theory of program tracing knowledge. My design focuses on helping people learn 1) basic PL knowledge and 2) program tracing skills; my prototype learning tool PLTutor focuses specifically on JavaScript. The central instructional design strategy in PLTutor is to build upon the experience of using a debugger, but 1) allow stepping forward and back in time, 2) allow stepping at an instruction-level rather than line-level granularity, and 3) interleave conceptual instruction and assessment throughout the program's execution.

Figure 3.1 shows the experience of using PLTutor to observe program execution over time. For example, Figure 3.1.a shows the path underlined in the first and second row of Table 3.1 for the token `0` in `x==0`, and Figure 3.1.b shows the third row in Table 3.1. The change to machine state displays on the next time step (see `0` on top of stack at Figure 3.1.c). This representation addresses the notational barrier to accessing the information in Table 3.1. As a side note, for brevity Figure 3.1 assumes `x` has value zero (e.g. at first time step).

The next two subsections discuss in detail how my reading-first spiral approach (partially implemented in PLTutor) teaches learners both the mapping from syntax to semantics and execution state, and program tracing.

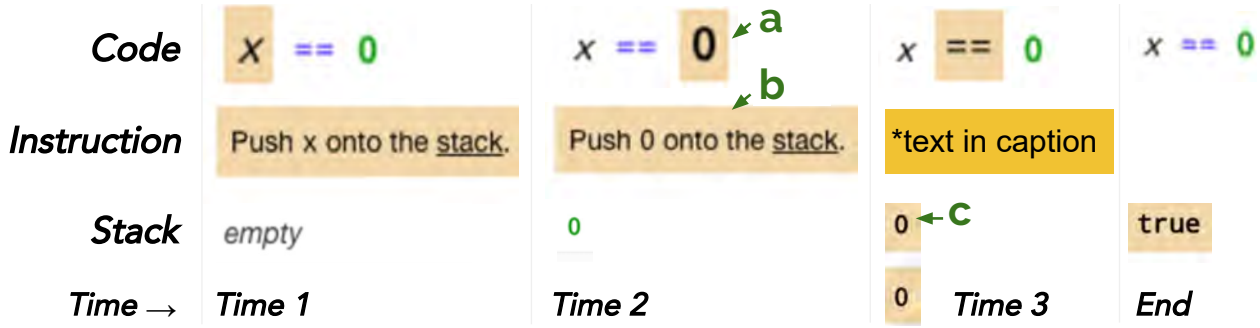


Figure 3.1: Stepping through three semantic paths covered by the example program `x==0`. *text is “Pop 0 and 0 off the stack, compute `0==0`, and push the result onto the stack".

3.2.1 Instructional Design for Basic Programming Language Knowledge

Now I theorize how learning may occur within PLTutor’s pedagogy. I start with an example in another domain, then return to learning PL semantics. I start from a theory of learning for “a causes b" relationships (also called causal inference). For instance, a child may see a switch flipped then a light goes on, and they may infer the switch caused the light to go on, just from one example. How can this rapid learning occur? A theory of causal inference identifies three key enablers for this learning: *ontology* (differentiating/recognizing entities and their causal types), *constraints* on relationships (how plausible are relationships), and *functional forms* of relationships (from how effects combine and compare, to specific forms like $f = ma$) [Griffiths and Tenenbaum, 2009]. For the light example, *ontology* includes a light and a switch as mechanical entities, *constraints* includes closeness in time (switches tend to control things quickly), and *functional form* may include knowing the switch up position activates (“turns on") whatever it controls. With these enablers, causal learning then occurs by *observing examples of the causal relationships* [Griffiths and Tenenbaum, 2009].

I theorize learning PL semantics as grasping the *causal relationships* from code to machine behavior. To enable causal learning, my key insight is to convert the abstract path representations from the prior section into a concrete causal visualization, showing the causal relationship between *tokens*, *machine instructions*, and the *machine state changes* caused by the instructions.

Now I summarize how my pedagogy provides the three key enablers for causal learning, for program semantics; afterwards I discuss them in a particular implementation (PLTutor). To provide *ontology*, I start learning with conceptual content describing entities (code, instruction, and machine model), show them in the visualization, and explain how to recognize them in the visualization. To provide *constraints* on relationships, I provide conceptual content emphasizing the mechanical relationship (to avoid the "computer as person" misconception, also known as the "superbug" [Pea, 1986]; causal learning theory gives us a new lens for why this misconception may be so damaging - having the wrong relationship misleads the causal learning process).[2] I also give *functional forms* via syntax highlights for the token part, text descriptions of the instruction part, and the change in machine model state over each small time step.

To create this causal model, I extract patterns from interpreter implementations. I introduce two abstractions—the *instruction* and the *machine model*—to simplify this knowledge, while retaining fidelity. Typically, paths through the interpreter end with manipulating the state of the program being executed (either by generating machine code or with PL statements (for an implementation hosted in another language)). I encode these state changes as one or more instructions for a machine model (which serves as a model notional machine [du Boulay et al., 1981]). The interpreter produces a list of instructions, which are then sent to the machine model. This separates the state of the program's execution entirely into the machine model. Using the machine instruction as a bridge between syntax and state transformation completes the path through the interpreter.

I can also connect instructions back to the code that causes them. These connections follow the semantic path, going back to the program code via the tokens that caused values in them. For example, for the path underlined in Table 3.1, the `PUSH 0` instruction connects to the `0` in the source code via the token Number(0); the `LOOKUP_AND_PUSH "x"` instruction connects to the x in Name(x).

---

[2]This superbug is actually directly taught as an anthropromorphic metaphor in early LOGO learning materials as the "little people" view of computations [Solomon et al., 2020]. At the same time in the LOGO culture thinking about ideas with metaphors and thinking about thinking was seen as more important - learning programming was an environment for learning epistemology and ways of thinking, rather than learning strict correct conceptions of PL semantics first.

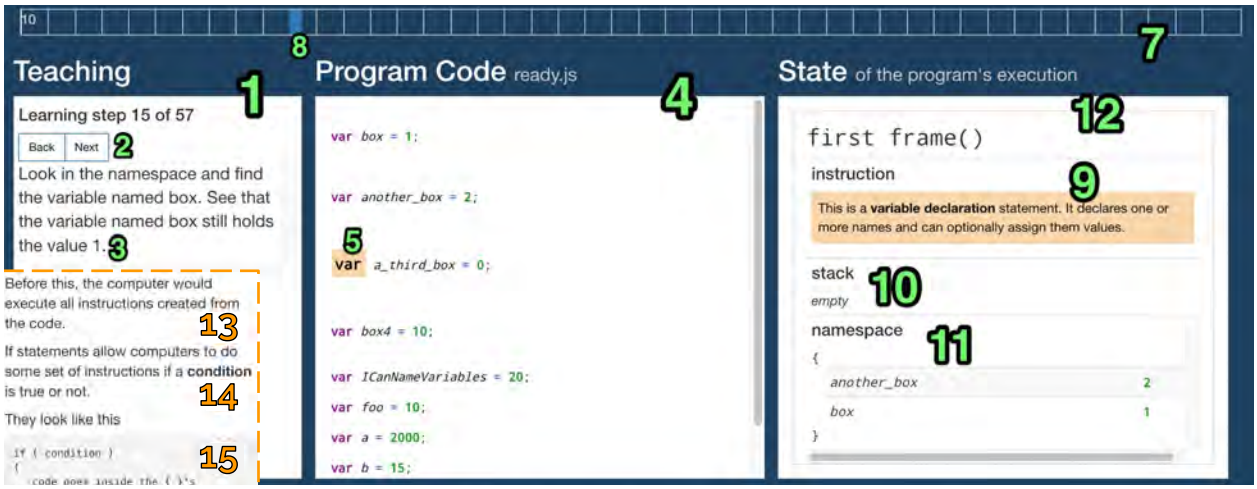


Figure 3.2: PLTutor showing an early lesson on variables: left, 1) the learning content and assessment area with 2) stepping buttons and 3) conceptual instruction; 4) program code with 5) token-level highlighting to show what caused the instruction. To show the machine model, we have 7) timeline of instructions executed for the program 8) current step, 9) current instruction's description, 10) stack, 11) namespace, 12) call frame. Lower left inset shows content for conceptual instruction for a later `if` lesson. 13) references relevant prior knowledge and contrasts what is new, 14) presents the goal of the construct, and 15) presents the syntactic pattern.

Figure 3.2 shows PLTutor, my web-based prototype that visualizes these causal connections. In this figure, PLTutor is visualizing the beginning of a JavaScript variable declaration statement, which is mapped to the *var* keyword in 3.2.5. The machine state is on the right with the machine instruction shown at 3.2.9.

In PLTutor, my pedagogy has learners *observe examples of the causal relationships* by stepping forward in time in the program's execution, one instruction at a time. A single step changes the token being highlighted, the current instruction, and machine state. For example, Figure 3.1 shows three steps for the JavaScript program `x == 0`. My prototype supports the full semantics of JavaScript, providing mappings between all constructs in the language and the machine instructions that govern JavaScript program execution.

Within this representation, my pedagogy identifies a unique causal role for the *instruction* between code and the machine model. The instruction *makes visible* the causal relationship between syntax and machine behavior. The instruction also provides *constraints* on relationships between code and machine state changes (a parser generates instructions from code, which is the only way code causes machine behavior). I also designed instructions to provide a set of *functional forms* that simplify and localize relationships; all instructions only 1) push onto an accumulator stack (either from the literals in the code or the namespace) 2) pop values from the stack, do a local operation only with those values, and push the result onto the stack, 3) set a value in the namespace, 4) pop a value and change the program counter (for conditionals, loops, and functions), or 5) clear the stack (which I map to the `;` token). These forms provide further *constraints* on relationships: for example, values from code tokens only change the stack, namespace values only come from the stack, and instructions only come from the code. These constraints and functional forms should help learners' causal inference [Griffiths and Tenenbaum, 2009].

Besides this causal role, the instruction also makes visible how a language executes syntax. Without it, stepping through the interpreted program line by line requires understanding how the computer navigates the program code notation. In contrast, stepping through a list of instructions only requires moving forward and backwards through an execution history and comprehending changes to machine state (see Figure 3.1).

PLTutor also conveys constraints, functional forms, and ontology by showing natural language explanations of the actions instructions are taking. PLTutor reinforces functional forms by showing an instruction's form as a description filled in with concrete values (see Figure 3.2.9, which explains how a variable declaration begins). PLTutor also shows learning content at 3.2.3 throughout the lesson.

To scaffold causal inference, the curriculum starts with *ontology*, with 5-10 minutes of conceptual instruction about the computer, code, state, the interpreter, instructions, and machine model. It provides ontology by describing the entities and how they are shown in the interface; for example, "The namespace is where variables are stored. This is like a table with two columns...". It also gives constraints on their relationships; for example, "In general, the list of instructions does not change as a computer executes a program." This information also serves as organizing concepts [Bransford et al., 2000].

3.2.2 Curriculum Design for Program Tracing

My theory of program tracing knowledge suggests a general approach for a reading-first pedagogy: show a faithful representation of each interpreter path, assess the learners' knowledge of each path, correct misconceptions, and cover all the paths for completeness.

3.2.2.1 Flexible teaching

To cover most of the paths, PLTutor uses a fixed, sequenced curriculum of example programs. Instead of stepping through execution steps directly as in prior work[3], PLTutor has a teaching layer of abstraction that decouples teaching a program from the program's execution. Each program has a list of *learning* steps, specifying 1) the learning content and/or assessment to show and 2) which execution step to show. This decoupling allows the curriculum to navigate anywhere in the program's execution when the learner advances to the next learning step.

There are three types of learning steps in PLTutor: *conceptual* steps (show conceptual instruc-

[3]Or sub-steps of each execution step with annotations to not show some steps [Sirkiä, 2018].

tional content), *execution* steps (show an instruction executing), and *assessment* steps (prompt a learner to fill in values, described shortly). Learners advance forward or backward through these steps by clicking the “Back" and “Next" buttons (Figure 3.2.2) or using keyboard arrow keys. Learners may also drag the bar (Figure 3.2.8) to scroll through execution steps. On the final learning step of a lesson, a “Next Program" button appears, which navigates to the next program in the curriculum. This contrasts with prior work in program visualization systems, in which learners must find and choose a next program for themselves, constructing their own curriculum from a menu [Sorva and Sirkia, 2010].

My pedagogy interleaves these three types of steps (conceptual, execution, and assessment) through a program’s execution.[4] To illustrate this, I describe the learning steps for a first lesson for a construct (such as `if`). These follow a pattern: reference relevant prior knowledge, contrast what is new (“Before this..." see Figure 3.2.13), present the goal of the construct (“If statements allow...", Figure 3.2.14), present the syntactic pattern (see 3.2.15), then scaffold learning strategies (“step through to see how it works", and, later, prompts for mental execution “What do you think this next if statement will do? Read through it and think, then step through it."). Where possible, steps introduce constructs with “equivalent" programs, based on similar instructions or state changes. For instance, in my arrays lesson, `is_day_free_0=false; is_day_free_1=true;` precedes `is_day_free = [false, true];` this may help transfer and provide *constraints* for causal inference.[5] After such code, steps show an example path and a low-level assessment (described shortly), then proactively address common misconceptions in turn by: 1) showing learning content against it, 2) executing a counter-example, 3) stepping through code with assessments for the misconception.

Besides the ordering of steps within each program, the stepping interface scaffolds perception of conceptual and execution information. On steps with learning content, learners experience a slight

[4]This is not a technical limitation, rather an instructional design choice to avoid having too many things change at the same time which overwhelms motion detection to perceive what has changed via peripheral vision.

[5]While this seemed like a not novel idea to me, I have not been able to find it described in the literature; I expect multiple teachers have invented this in different contexts but not recognized it as an instructional principle to be consistently implemented throughout a curriculum (and natural language explanations of semantics are often seen as sufficient to teach PL semantics).

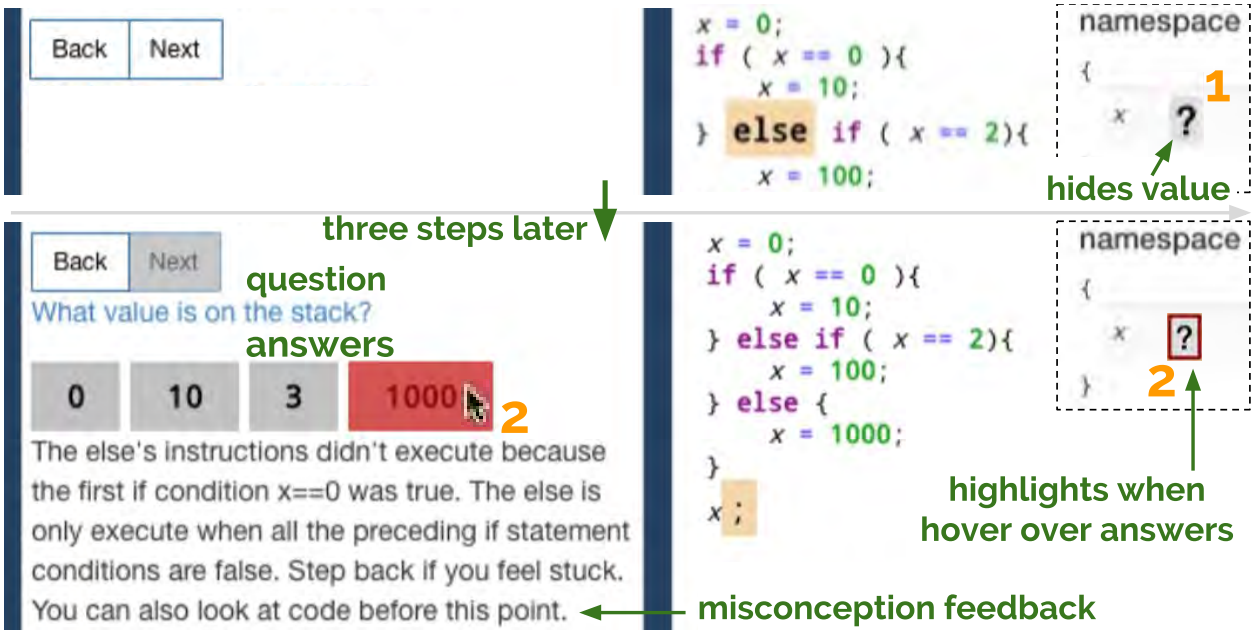


Figure 3.3: Assessments scaffold state, hiding values with a ? (see 1), so learners mentally execute semantics to answer. The assessment shows three steps later. This allows assessing from the step to multiple line or program level granularity, without requiring navigation restrictions to hide values.

1-second pause before they can advance, to encourage reading. At key points in the middle of a program's execution, learning steps stop advancing execution to show conceptual instruction. For example, PLTutor pauses in the middle of a condition evaluation to describe what is happening when first learning `if` statements. Execution steps show causality temporally, and having many steps shows many examples of the causal relationships, as recommended by [Griffiths and Tenenbaum, 2009]. Their granular, sub-expression level of detail may help structure inferences from these lower level steps into higher level inferences across multiple steps and lines of code [Griffiths and Tenenbaum, 2009].

3.2.2.2 Scaffolded Assessment at Multiple Levels of Granularity

To aid higher-level causal inferences, PLTutor assesses knowledge of each path at multiple levels. For the single execution step level, it navigates to a step and hides a value for the learner to fill in. Figure 3.3 shows an assessment across multiple steps. Using value hiding and the learning step's

ability to control what execution step is shown, this enables scaffolded assessments, showing some instruction or code navigation, across multiple levels of granularity, including the simple effects of one execution step (done in prior work [Sorva and Sirkiä, 2011]) to showing the resulting state of many execution steps as if it were one large step.

PLTutor also scaffolds links between assessment question phrases and machine state by showing which question to fill in; hovering over any answer shows a box around the corresponding value in the machine state visualization (see Figure 3.3.2). Assessments also shows misconception feedback for inaccurate answers (see bottom Figure 3.3).

3.2.2.3 Example lesson

To illustrate the instructional design, let's follow Sally, a learner who has already gone through example programs showing paths for variables, arrays, operators, and booleans. She is on the first program for the `if` statement (the corresponding lesson is shown in Appendix B on page 284). This lesson covers two paths: `if` with true condition and `if` with false condition, both without `else` clauses.

For `if` statements, there is no prior equivalent semantic path, so the learning content contrasts with prior rules using conceptual instruction (for example, see inset lower left in Fig. 3.2). There is a slight 1-second pause before she can advance past this step (and all steps with learning content), to scaffold perceiving and reading them. She then steps forward through steps that advance the execution step shown. She sees the highlighted code tokens (see 5 in Fig. 3.2) that cause the current instruction (and show the AST relationship for the syntax-semantics-state mapping). After several steps she reaches the first if statement. Here the learning steps stop advancing execution to show three steps of conceptual instruction: "The condition is what we call the expression inside of the parentheses that follow the if. Here the condition is 10 > 0" "If the condition leaves true on the stack, the computer executes the code within the curly braces { }." "Read the instruction descriptions and step through this carefully." The learning steps then continue advancing execution. Sally steps through the if statement's execution, then continues to the next if, where she sees a low-level assessment on the condition evaluation step (because she needs to know the condition

evaluated to false to understand the cause of skipping over the body of the if statement). She must answer accurately to proceed (but can always step back). She answers false correctly, then gets a step of conceptual instruction: "**Because the condition left false on the stack** (it was 0 > 10), the computer **skips over the instructions** from code between the next { }. It does this by changing the next instruction to execute. Watch it skip over." The next learning step shows execution advance, jumping over the if body, then Sally sees conceptual instruction: "Woah, the computer just skipped over the code inside the { }. That's the first time you've seen that happen. Thus, the value in x didn't change. It is still 0.". Sally continues stepping through learning steps and answering assessments until the last learning step, then clicks next program to advance in the curriculum.

3.2.2.4 Meaningful programs and PL community values

My reading-first curriculum also spirals upwards from teaching and practice on small granular parts of programs to larger meaningful programs; this instructional and curriculum design draws on social theories of learning. In addition to introduction and practice lessons, the curriculum design includes review lessons that cover programs with meaningful social uses. These review lessons apply constructs together and occur after the end of operators, conditionals, and the loops material. These lessons describe how constructs can be used with each other. They contrast larger "equivalent" code segments, justifying language features by appealing to "good" properties of code like readability, brevity, and ability to modify or reuse; this and other conceptual instruction aims to teach the values of the programming community.[6] These integration lessons are designed to increase retention and motivate learning by showing how constructs are used together for actual problems [Bransford et al., 2000], and to connect knowledge to the design goals of the programming language.

The tone and writing of the learning content is also important, designed to mimic being introduced to the programming community of practice by a sympathetic old-timer [Lave and Wenger, 1991]. The voice of the tutor speaks directly to the learner, like an expert was sitting down next to

[6] Again, while this seems like a not novel idea to me, I have not been able to find it described in the literature as an instructional principle to be consistently implemented throughout a curriculum.

them walking through programs in a debugger while explaining them. The learning content is sympathetic to the learner, basically apologizing for the language design choices made by Javascript's designer; here are two examples: "Here are two more boolean operators that are very important but the language designers chose to use some strange symbols to represent them. The AND operator && ..." and "Like an if statement, it uses the *condition* expression to decide to keep going or not. We think the word repeat instead of while would have been more clear for people, but we didn't design this language.". This injects some humor about how we are seemingly stuck with weird languages like Javascript, with all our internet browsers running a language that was designed in about two weeks. The voice of the tutor also suggests the learner might make a better language some day throughout the tutorial; for example, "Some of those [language design] decisions will seem confusing and mysterious at times, so *we'll try to understand why they were made*. This can help us remember how they work or empower us to create better languages."

3.2.2.5 PLTutor Limitations

As a prototype, PLTutor only partially implements the principles I have discussed for a reading-first pedagogy. It covers all the paths in my JavaScript interpreter except for strings, I/O, objects, some unary and binary operators (like - and modulus), and error paths (like invalid variable names or syntax errors). For example, PLTutor does not show the failing examples required to fully specify variable naming patterns. While ideally a reading-first curriculum should cover these eventually, for this initial evaluation I expected later writing pedagogy to cover them, and language runtimes make them visible with error messages. See Appendix B for the entire curriculum that was implemented. The curriculum was not optimized, including the perhaps overly long conceptual introduction before the first executing program.

While PLTutor's assessments directly or indirectly cover much of each semantic path, it leaves some out and does not fade scaffolding entirely. It directly assesses machine state changes by having users fill in values in the machine state via linked assessments. It indirectly assesses control flow via simple value changes (which depend on the variable assignment path in earlier lessons); Figure 3.3 shows one. PLTutor does not assess ontology directly or fully remove scaffolding in its curriculum;

for example, it always shows machine state, instruction execution steps, and allows stepping.

Finally, at the time of my evaluation, PLTutor was very much a research prototype. When I evaluated it, it had usability issues and the environment made little effort to engage learners, introducing numerous barriers to sustained engagement and learning. The choice of the stack machine visualization also may have over-complicated learning compared to a term rewriting notional machine, but for the purposes of the causal learning design I wanted to try a clearly mechanical model for the initial evaluation.[7]

3.3 Evaluation

What effects does a theoretically-informed, reading-first curriculum with flexible teaching granularity have on learning code reading and writing skills? To investigate, I conducted a formative, block-randomized, between-subjects study comparing a reading-first tutorial, PLTutor, with Codecademy [Codecademy, 2016]. I chose Codecademy for its traditional writing-focused spiral pedagogy [Shneiderman, 1977] and quality from four years of curriculum refinement. I label the PLTutor condition *PLT* and Codecademy as *CC*.

3.3.1 Method

The study inclusion criteria were undergraduates that had not completed a CS1 course in college and had not used a Codecademy tutorial. I recruited students starting a CS1 class that followed a procedures-first spiral writing pedagogy using Java [Reges, 2006]. Participants came to a Saturday 10:30am–6pm workshop, took a pre-survey and a pre-test, used a tutorial for 4.33 hours, and then took a post-test and post-survey. As a pre/post-test I used the SCS1 [Parker and Guzdial, 2016, Elliott Tew, 2010], a measure of CS1 knowledge with a validity argument. Both surveys used validated measures for fixed vs. growth mindset [Blackwell et al., 2007] and programming

[7] Other notional machine designs such as term rewriting also have some downsides, such as causing the text of the program to change as it executes, which can present a barrier for transfer to mentally executing code in an editor during program writing (people tend to cling to surface features of representation when developing mental models [Schumacher and Czerwinski, 1992]). When looking at a program in an editor the text surface does not change and keep track of evaluation progress for the learner.

self-efficacy [Ramalingam and Wiedenbeck, 1999]. The pre-survey also measured daytime/chronic sleepiness using the Epworth Sleepiness Scale [Bailes et al., 2006], as prior work argues these affect learning [Hochanadel and Finamore, 2015, Murphy and Thomas, 2008, Ramalingam et al., 2004, Curcio et al., 2006, Dewald et al., 2010].

At the first two lectures of the class and via email, I advertised the study as a chance to excel in the class, potentially biasing towards motivated and at-risk students. Overall, 200 of 988 students responded to an emailed recruitment survey and 90 met our inclusion criteria. Using this survey, I block randomized [Hinkelmann and Kempthorne, 2008] participants into a condition using hours of prior programming, self-reported likelihood of attendance, age, and gender, then invited subjects. From these blocks, I randomly invited participants from each block, ultimately having 41 attend the workshops. After confirming attendance by email, I sent subjects the room for their condition.

The two instructors of the one day workshops followed a written experimental protocol and coordinated to make any necessary day-of changes jointly. They then showed an introductory SCS1 test directions video, gave a 1 hour pre-test, then showed video instructions for their condition's tutorial and stated students would have 4 hours and 20 minutes to learn the material. They served lunch during the tutorial period at 1PM. After a 10-minute break following the tutorial, students had 1 hour for the SCS1 post-test and could start the post-survey when done.

For the purposes of reproducibility, here are some further details about what the instructors said. After the pre-test, they read "Thank you for taking the pre-test. It's something we have to do as part of the study and we expected the vast majority of you to not know anything that was on the test. Your performance on the test has no relationship with your ability to learn today. Next, we'll try to learn some of the material on that test today, after the break." There was a short 5-minute break before students watched the video instructions for their tutorial. After the video the instructors said "You will have 4 hours and 20 minutes to learn the material. You will not be able to take the post-test early, so if you finish early, I encourage you to review the content. You may now begin using <tutorial name>. Please be sure to take breaks as necessary." After the post-test the instructors said: "Thank you for taking the post-test. This test is very challenging for people, even people that have completed both <CS1 coursename>and <CS2 coursename>. Your performance on the test has

no relationship with your ability to learn, and does not indicate anything about your ability to learn in <CS1 coursename>. This is especially true if your native language is not English. If anything, the test measures how well the tool was able to teach you."

I operationalized learning outcomes by proxying program reading with SCS1 score and writing skills with midterm grade. Learning gain (posttest score−pretest score) is noisy because it combines pre and post test measurement error [Bonate, 2000]; I also counted per-question and per-individual performance from incorrect on pre-test to correct on post-test (which I call *FT*, for false-to-true), as well as likely prior knowledge as correctly answering a question on both the pre and post test (*TT*, for true-to-true). I defined *learning capacity* for a person as (the # of questions not left blank)-*TT* and *learning capacity* for each question as (# of people that did not leave the question blank)-*TT*. I defined *LCL*, the % learned that could learn, as *FT* / learning capacity (like normalized gain in [Taylor et al., 2014]).

The SCS1 system randomly lost some tests; I dropped those participants, reducing sample size to 18 in PLT and 19 in CC. I separated novices (operationalized as less than 10 hours of self-reported experience and no prior CS class) from experienced students (had prior CS class or >10 hours of self-reported experience).

3.3.2 Results

Despite random assignment, I found differences that may have confounded measures of learning gains. I analyzed these and other differences by default with the Wilcoxon rank-sum test for non-normal data. I add and note a t-test when Shapiro-Wilks's normality test had $p > 0.1$ for each group (still has low power for my sample size). Pre-test SCS1 differences between the two conditions were large and marginally significant (CC–PLT mean=1.65, W $p < .111$, t-test $p < 0.058$), but self-reported prior programming experience and mindset did not differ significantly.

While many individuals in both groups achieved higher SCS1 scores, comparing *within* each condition, only PLT's post-scores were significantly different from its pre-scores (p<.0044) (for CC p<.089). Figure 3.4 shows descriptive statistics; for comparison, students near the end of a CS1 course score from 2 to 20, $mean = 9.68, SD = 3.5$ [Parker and Guzdial, 2016]. Comparing

conditions, PLT had higher individual *FT*, with Cohen's d=.59 (p<.12, t-test p<.075); for *learning gain*: Cohen's d=.398 (p<.41, t-test p<.24); for the % learned that each person could learn (*LCL*): d=.34 (p<.39, t-test p<.31). To control my analysis for other variables, I tried to fit post-score with linear and binomial generalized linear models, but residuals strongly violated modeling assumptions.

For specific questions in the SCS1, PLT outperformed CC on 37% of questions and CC outperformed PLT on 22%, based on between group difference of $\geqslant .1$ in LCL (the % of people who got the question right that did not already get that question right on the pre-test). Figure 3.5 shows questions sorted from left to right by this LCL difference, with * at bottom for non-overlap of their 95% confidence intervals of the probability p of a binomial distribution estimated from x=FT and n=capacity (Wilson [Brown et al., 2001]).

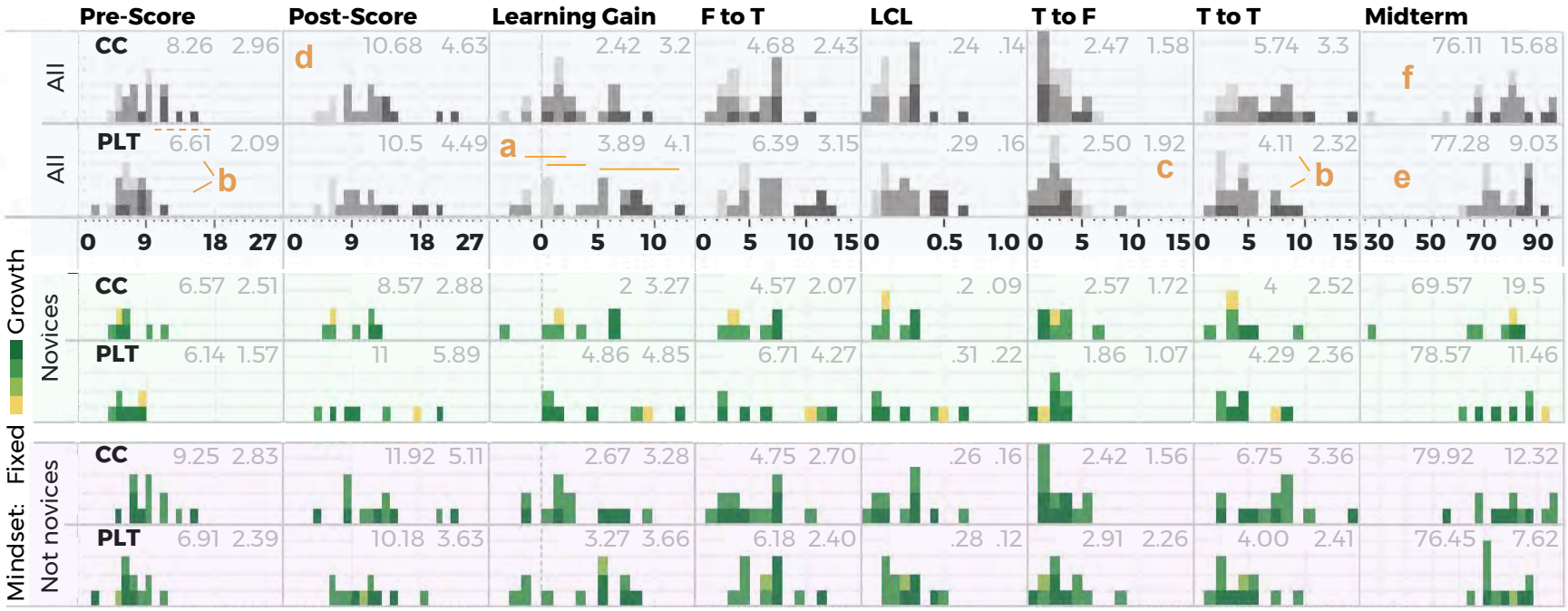


Figure 3.4: Histograms (with mean then SD inside) by condition (top) colored by *post-score* (dark grey: $\geqslant 13$ (about a SD above the mean of students finishing CS1 from [Parker and Guzdial, 2016], grey: within a SD, light grey for below), then experience (mindset color).

CC did better for some code completion (Q18,25,21) and writing conceptual questions (e.g., Q1: When would you not write a for loop... Q6: Imagine writing code for each task - can you do it without conditional operators...). It also did better on topics missing from PLT's curriculum like strings (Q18) and a tracing question with modulus (Q19). In contrast, PLT did better for tracing (Q2,23,3,12,24,14,8), a tracing conceptual question relying on sub-expression detail (Q10), and complex code completion questions (Q13,Q26). PLT also did better on topics missing from CC's curriculum like recursion (Q14 and even Q24 involving strings). PLT also did better on a tracing question with strings that only used array syntax and semantics (Q15).

PLT had less variation and a more normal distribution for later writing outcomes compared to CC (see Figure 3.4.f, 3.4.e). Shapiro-Wilks normality test rejected CC's midterm distribution ($p<.013$) but not PLT's ($p=.4723$). For PLT, midterm fit a linear model on each of: *learning gain*: (adj-R^2=.6469 (95% CI: .28, .84 by [Kelley, 2007a, Kelley, 2007b]) $p<.00004$ residual SE=5.37), *post-score*: (adj-R^2=.5502 $p<.00025$), *LCL*: (adj-R^2=.53 $p<.00037$), and *FT*: (adj-R^2=.4983 $p<.00064$). For CC, only a post-score model had significance (adj-R^2=0.1784, $p<.041$, SE 14.21). PLT had no midterm failures (vs. 2 in CC). Midterm average did not differ significantly between PLT or CC (see 3.4.e), or those in the recruitment group that met the inclusion criteria and did not participate, midterm $m = 72.1, SD = 22.1$ (n=38).

To partly check the validity of midterm as a proxy for writing skill, I offered $6 for a photo of the midterm (with per question grades) to those that met the inclusion criteria (90) and got 17. A linear model from total score predicted the writing part (adj-R^2=.85) much better than the other parts (adj-R^2=.61), suggesting total midterm score varies fairly closely with the writing portion.

As a manipulation check, two months after the workshops I offered post-midterm tutoring. In each session, before tutoring, I conducted a think-aloud interview for tracing mental model granularity by prompting learners to "Underline the code as the computer sees it, then describe what the computer does" for Java versions of program B2 & G3 from [Sleeman et al., 1986]. Learning gains and midterms mostly increased with more sub-expression tracing (except P5). Two participants (P1 and P2) responded from CC. P1 had 2 TT, 2 FT, and a 54 midterm; her tracing model had some sub-expression (but not for control structures and had an early loop exit error); P2

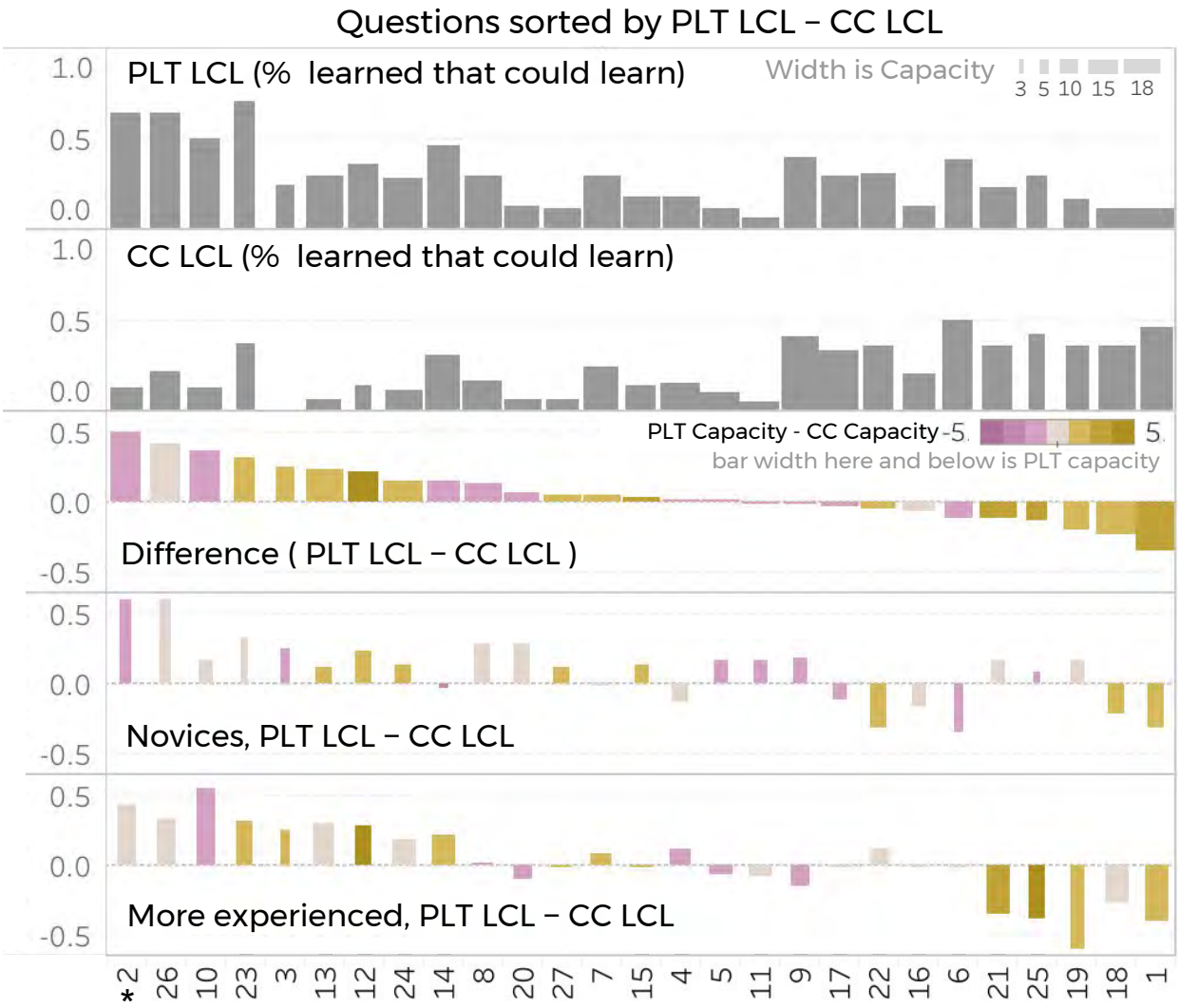


Figure 3.5: Per-question statistics, ordered by PLT's LCL minus CC's LCL. Bar width shows capacity.

(2 TT, 6 FT, 79 midterm) was mostly line-level but separated assignment (like `y=1+1;`) into 3 steps: the left side `y`, then =, then the rest `1+1;`.

Four PLT participants (P3 to P6) responded. Compared to P2, two showed more but non-uniform sub-expression with self-caught misconceptions, P3 (4 TT, 7 FT, 72 midterm) and P4 (5 TT,7 FT, 77 midterm). P5 (8 TT, 12 FT, 86 midterm) had a line level model. P6 had a consistent sub-expression model (3 TT, 11 FT, 85 midterm).

I also asked which tutorial features they remembered, as a gross check on importance or causality. Better learning outcomes mostly increased with more correct and complete feature recall, suggesting they impacted learning. In CC P1 and P2 both recalled writing, exercise feedback, and help; only P2 had the print output. In PLT, P3 recalled learning content and (incorrectly) writing code only, with the stack shown briefly during execution with no stepping controls; the others recalled content, stepping and assessments; for the state display (see right side of Figure 3.2), P4 had namespace (3.2.11), P5 had steps bar (3.2.8), P6 had all except instruction (3.2.9).

3.4 Threats to Validity

While I made efforts to ensure validity (minimizing confounds, block-randomizing group assignment, measuring confounding factors, avoiding early-riser effects [Dewald et al., 2010], mitigating experimenter bias, etc.), there are still several threats to validity.

Differences between my study and the validity studies of the SCS1 complicate the interpretation of my results. I post-tested within 4.3 hours of the pre-test; a carry-over effect may inflate post-scores (e.g. remembering questions) [Allen and Yen, 2001]. Guessing, especially by novices, may have impacted scores. While the SCS1 is the best publicly available measure, its validity arguments do not formally generalize i) to novice test takers, ii) a pretest-posttest context, and iii) as a measure of learning gains. My measure of writing skills (the midterm) had unknown measurement error and lacks validity argument.

Motivation differences, participant fatigue, measurement error, unmeasured participant variation, and differences in workshop setting also threaten validity. Internally, instructor variations may favor the Codecademy condition, which had a more experienced teacher. The PLTutor instructor also

had to leave the room for 45 minutes to handle a lunch issue. Externally, the study's short duration may create a ceiling effect on learning gains. The study protocol, curricular quality, time-on-task, program domain, and pedagogical and lack of adaptive tutorial features affect results. In particular, in informal interviews, participants reported frustration with repetitive practice in the PLTutor curriculum, which may have reduced engagement and therefore learning.

3.5 Discussion

I have presented a new theoretical account of program tracing knowledge, a new pedagogy for teaching this knowledge embodied by PLTutor, and empirical evidence of the effects of PLTutor on program tracing and writing skills. These effects included:

- Higher total and question-specific learning gain than Codecademy (overall 37% of questions and 70% for novices).

- Less midterm variation and no failure on the midterm. Learning gains from the tutorial also strongly predicted the midterm, suggesting a strong relationship or shared factors between learning rate in the tutorial and the class.

- More learners who started with low pre-scores had large learning gains (see dark grey in pre-score and *TT* (likely prior knowledge) in Figure 3.4.b).

My study suggests greater learning gains for PLTutor compared to Codecademy. PLTutor matched Codecademy's post-scores even with a significant initial deficit (see Figure 3.4.b). This might just be mean-reversion for Codecademy (guessing on the pre-test with less luck on the post) but true to false shows little to no difference (see 3.4.c). The other interpretation is that PLTutor brought its less experienced group to parity with Codecademy (see 3.4.d); if the writing tutorial was better at teaching program tracing, it ought to have magnified initial differences. PLTutor also had more learning at the question level, doing better on 37% vs. 22% for Codecademy (1.68 times more). Question-level differences might come from sampling error, which is hard to model without item

response theory parameters for the SCS1. However, these differences always aligned with curricular differences (no recursion in CC, no strings or modulus in PLTutor) and theoretical explanations—for example, writing did better on 3 out of 9 code completion problems, as did PLTutor for 8 out of 13 tracing problems. This supports the interpretation that my results do not just come from noise or guessing differences, though my small study still has threats to validity.

My empirically strongest result is that PLTutor normalized midterm outcomes. PLTutor had no failures vs. 2 in CC (see 3.4.f vs. 3.4.e). With only one early measurement, in my small study only PLTutor learning gains predicted midterm (adj-R^2=.64), among the best of work predicting CS1 outcomes (adj-R^2 .44 to .46 [Wilson and Shrock, 2001, Watson et al., 2014, Mayer, 1985]), better even than those using mid-course measures like homework or self-efficacy (adj-R^2 .35, .58, .61) [Lishinski et al., 2016, Krapp et al., 1992, Watson et al., 2014]). In pre-score in Figure 3.4.b, PLTutor also has more dark high post-scorers coming from lower scores vs. CC. This improved equity in outcomes may help scale learning in diverse populations. Future work should confirm this pattern and see if normalizing has the downside of reducing outlier high outcomes (compare far right of 3.4.e & 3.4.f).

I also saw learning gains comparable to full quarter or semester long courses with both tutorials in ∼4 hours (see Figure 3.4.a), similar to prior work [Lee and Ko, 2015], yet unexplained by prior CS knowledge or traits I measured. Some gains varied from losing to gaining 2-3 points on the post-test, perhaps guessing noise. However, my participants' post-test distribution looked similar to those near the end of CS1 in [Parker and Guzdial, 2016]; this was either genuine learning, recruiting bias that skewed my sample towards more motivated or at-risk students, or test-retest carry-over score inflation. Even extreme learning was not uncommon; in ∼4 hours, in PLTutor 4 learners (22%) moved from a below average pre-score to nearly above a SD of [Parker and Guzdial, 2016]'s mean (3 (16%) above 13.18), and one from a score of 8 to 20, the maximum from 189 students in [Parker and Guzdial, 2016] (in CC, one 8 to 14 and an 11 to 20 also). In PLT these outcomes continue in the midterm (see dark at 3.4.e).

What skills or knowledge explains such fast learning without prior domain knowledge? Can we teach it and dramatically improve CS1 and even other CS education? Most learning theories

frame learning as hard and time consuming, and transfer as fragile; they poorly explain these results. In contrast, causal inference theory says learning is facile and transfer instantaneous with the right conditions. I applied this theory in my pedagogy design and saw large gains, making it a promising direction. Future work may search for factors that lead to rapid learning by measuring learners' prior knowledge or traits then analyzing learning outcomes only, or jointly change the design of pedagogy or tools used, in an attempt to either increase or reduce extreme learning gains.

Decades of studies have attempted to improve outcomes for learning programming; I found something one can measure in only 8 hours (learning gain from PLTutor) which is highly predictive of long-term outcomes. We might be able to use this (or other good predictors) to improve the rate of experimentation and discovery, going from 3-4 studies per year using course outcomes to one per day using a proxy (if larger studies confirm their predictive ability).

Future work should investigate tools and curricula based on comprehension-oriented strategies, especially given the comparative lack of exploration and positive early results (mine and others like [Hertz and Jump, 2013]). PLTutor had as good or better overall performance compared to a mature writing-oriented tutorial created with millions in funding. It seems unlikely PLTutor, with almost no curriculum experimentation, has found the ceiling for comprehension tutorials or pedagogy.

Chapter 4

A FORMATIVE ASSESSMENT OF PROGRAM TRACING SKILLS

In the last section I developed a theory of basic PL knowledge as mappings from token-level syntax to semantics and execution state, and used that theory to design a new reading-first curricular approach and interactive textbook. In this section, I extend that theory to include nested combinations of syntax to semantics mappings. Based on those combinations, I generate a systematic assessment of program tracing skills. I draw on Kane's educational assessment validity framework to make an argument for validity for using the test to target feedback for learning, supported by an empirical study of that specific use. This work[1] was peer-reviewed and published in the ACM Koli Calling International Conference on Computing Education Research (with an initial analytical validity argument; I will submit the chapter's empirical study in a future paper) [Nelson et al., 2019].

4.1 Introduction

Formative assessments can help improve learning when they provide actionable information for better targeted instruction, practice, and feedback [Kingston and Nash, 2011, Dunn and Mulvenon, 2009]. Decades of evidence show formative feedback has one of the best effect sizes for learning interventions, according to a review of 500 meta-analyses [Hattie, 1999, Hattie and Timperley, 2007] and another review [Kluger and DeNisi, 1996]. Within computing education, validated formative assessments may help in learning foundational skills, such as program tracing, program writing, problem-solving, and other computing skills [Xie et al., 2019b]. Students in computing also prefer formative assessment, looking for "assessment as guidance and opportunity to learn" [Riese, 2017].

[1] I led this work but also had wonderful collaborators. Amy advised the work, and connected with Min Li and Matthew Davidson, who introduced us to Kane's framework for assessment validity, and were very helpful with orienting to the literature in that area. Andrew Hu did undergraduate research as part of the study, conducting a qualitative test session and served as a second rater for the qualitative coding in the study. Benji Xie helped at the end of the writing stage as a co-author and fresh pair of eyes to make the work more understandable.

Computing education research (CER) has strong initial work creating assessments with validity evidence for measurement of a construct, such as self-efficacy [Danielsiek et al., 2017] or CS1 knowledge [Tew and Guzdial, 2010], but the opportunity to design or evaluate assessments for validity specifically for formative use remains unexplored (as well as formative assessment for program tracing skills). Decker and McGill catalogued instruments used or published between 2012-2016 by searching online and reviewing the table of contents for ICER and the CSE and TOCE journals. They found 13 instruments for knowledge of computing or computer science, 3 for CS1, 1 for CS2, 6 for computational thinking, and 3 for advanced skills; and 31 non-knowledge instruments that "...measured constructs such as self-efficacy, anxiety, confidence, enjoyment, sense of belonging, intent to persist, and perceptions" [Decker and McGill, 2019]. Margulieux et al. reviewed measurement practices in CER papers reporting qualitative and quantitative human-subjects studies from 2013-2017 and found 16 standardized computing-specific measurements used. None of these present arguments for validity for formative use of these measures.

Making a formative assessment requires precise questions. While summative and formative assessments can have similar items, formative assessments must be much more carefully designed to diagnose what learners do and do not know. For example, a formative assessment that asked learners to write programs might fail to precisely identify what learners do and do not know about prerequisite skills, such as a programming language's syntax and semantics [Xie et al., 2019b]. And in fact, prior work shows many course exams and even validated assessments of programming knowledge have items that require 10-20 concepts to answer correctly, even for basic skills like program tracing [Luxton-Reilly and Petersen, 2017, Luxton-Reilly et al., 2018a, Petersen et al., 2011, Tew and Guzdial, 2010]. This lack of granularity can result in learners successfully learning several aspects of a programming language's semantics, but then getting a zero on an exam because they do not understand a single operator (as occurred with modulus operators in [Kennedy and Kraemer, 2018]). Moreover, if such assessments produce low scores that do not reflect learning progress, a learner might feel confused, demotivated, and even come to believe they lack sufficient ability in programming to continue studying it.

This chapter applies Kane's validity framework (described in the Related Work in Section 2.2.2)

to the design of formative assessment of program tracing skills. This chapter describes a model of program tracing skills (Section 4.2.1), then contributes a fine-grained scoring model for part of it (Section 4.2.2), the design of granular items to precisely observe tracing skills (Section 4.2.3), a test design that samples from a space of possible items (Section 4.2.4), a feasibility argument for formative use of the test (Section 4.2.5), and an empirical evaluation supporting formative use (Section 4.3). These contributions provide an exemplar for the CER community's continued and critical work on CS assessments, building upon advances in educational measurement.

4.2 A Program Tracing Assessment for Formative Use

Kane's framework for a validity argument can help researchers better design assessments by considering their strengths and weaknesses for a specific use (see Section 2.2.2). This section uses Kane's framework to design a formative program tracing assessment and provide theory-based validity arguments as design rationale, by discussing five aspects of the assessment design: a model of how tracing is performed in non-test contexts, how to model test performance with a *fine-grained scoring* model using nested combinations, how to design *granular* items suitable for scoring, how to compose those items into a test, and how to score the test and use scores for formative use.

4.2.1 Tracing Performance Model

Within Kane's framework [Kane, 2013], a key requirement for developing a validity argument for a formative assessment is a "model of the target domain." For program tracing, that means having a theory of what *causes* both novice and skilled tracing performance in contexts outside test tasks. This model helped me design an assessment, but also understand the limitations of the design and how to evaluate it.

My model of the target domain of program tracing builds upon the theory of basic PL knowldge from Section 3.1, which proposes that program tracing performance by any learner is ultimately caused by some set of beliefs about how specific programming language (PL) constructs execute (in PL terms, beliefs about their semantics). For example, learners skilled in program tracing likely

have beliefs that closely mirror a PL's *actual* semantics; for simple `if` statements without an `else` branch, this would mean having the ability to precisely follow such a conditional's two execution paths (the true and false branches through and around the *then* block). Less skilled behavior may have different beliefs about semantics, which may lead learners to execute a conditional differently (and incorrectly), such as *always* executing the *then* block, independent of the condition.

Beyond beliefs about a PL, I also argue that tracing performance is mediated by other skills, using prior work. For example, *perception* matters because if a learner misperceives an operator (e.g., > instead of <), they may trace incorrectly, even if they know the correct semantics. Similarly, *strategic* skills matter: many studies document how learners sketch and mark paper questions to keep track of values during tracing [Cunningham et al., 2017]; teaching strategies for distributing cognition in this way helps people avoid errors [Xie et al., 2018]. As Schulte's Block Model describes, some learners may use higher level program comprehension strategies, like detecting idioms or inferring semantics from variable names and context, to avoid tracing code [Schulte et al., 2010]. Finally, learners may simply guess in a test setting.

Of course, the brief summary above does not fully describe the rich complexity of program tracing skills. It does, however, illustrate the challenge of accurately measuring tracing ability at a level of granularity sufficient for guiding learning. It also illustrates the kind of domain nuances necessary for strong validity argument.

4.2.2 Scoring Model

Kane's framework suggests the test needs a *fine-grained* scoring model for formative use. Achieving this requires requires some simplification of the complexities in the domain of tracing. Here I describe the simplifications made to model tracing skills while retaining granularity needed for targeting weak skills with remedial instruction and practice. This constitutes the foundation of my *scoring* argument.

First, accounting for all factors in the previous Section 4.2.1 would require a very sophisticated and complex assessment that covers both basic PL knowledge, strategic knowledge, and many other factors. In this exploratory work, I simplified this by focusing on PL semantics knowledge, by

not explicitly modeling tracing strategy, code perception skills, meta-cognition, or other response processes (like guessing). I partly address these simplifications in my item and test design in later sections.

Given this decision to only model basic PL knowledge, I next needed to decide how to model this knowledge; my choices here require a more extended justification and discussion. Novice understanding of a PL's semantics may not align completely with a PL's actual semantics. Novices can have brittle models of a particular language construct's execution, but they can also believe that language constructs have dependencies that the PL actually does not have (e.g., assuming assignments to local variables change aliased global variables). If the scoring model cannot represent these varying conceptions of semantics, it will not be able to guide the specific feedback needed to refine these conceptions.

To model varying conceptions of semantics, I began with the observation that if a learner was entirely free of misconceptions, their PL semantics knowledge *could* be precisely modeled as what semantics they can and cannot accurately mentally simulate. This might be true for people who have mastered a programming language, but is certainly not true for everyone else: prior work has identified misconceptions including *imagined* interactions between the semantics of different constructs [Pea, 1986, Perkins and Martin, 1985]. For example, a novice may understand paths involved in a function that calls another function, but trace incorrectly when a function calls itself, even though the PL's semantics makes no distinction of such cases. Thus, I could not model that conception if the scoring model only tries to model knowledge of individual language construct semantics.

A much more fine-grained scoring model for tracing skills is to model a learner's ability to accurately trace *every possible* concrete program that a PL could express. This highly granular approach is infeasible, as the space of possible programs is infinite.

Instead, I chose a middle ground, which attempts to achieve *sufficient* granularity by modeling prevalent *compositions* of individual PL constructs, and modeling learners' ability to correctly trace all possible executions of those compositions. For example, rather than just modeling knowledge of conditionals in isolation, I modeled knowledge of conditionals nested inside of conditionals (and all

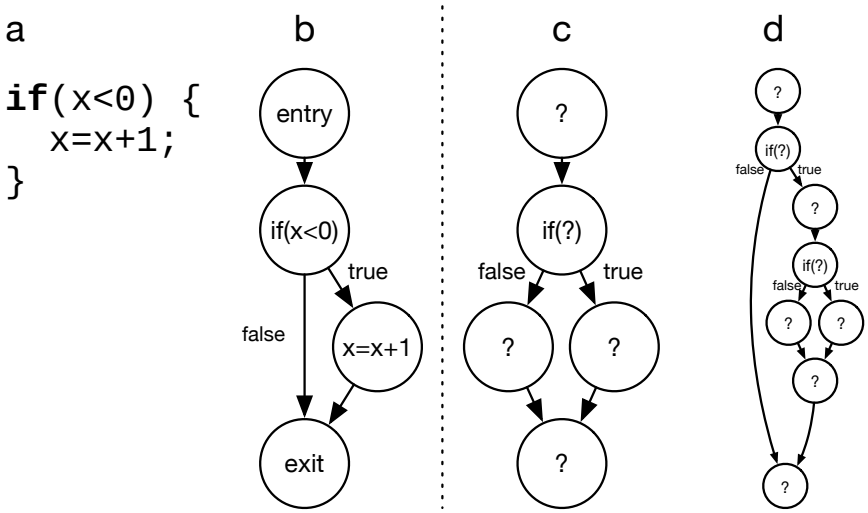


Figure 4.1: At label (a) a concrete program, then (b) its control flow graph (CFG), with specific conditions and assignments. At (c) a *semantic* CFG, (d) a *composite* semantic CFG, with ? placeholders.

their possible execution paths), or conditionals inside of loops (and all their possible executions paths). For example, `if(...){ if(...){...} }` has three possible execution paths: the first `if` executes but not the second `if`, both execute, or neither executes.

To explain this more precisely, I drew on two representations of program execution from program analysis. I represented PL construct semantics as *control flow graphs* (CFGs) to capture the possible executions of an individual construct, and *data flow graphs* (DFGs) to capture dependencies between uses and definitions of variables. A CFG represents potential executions through AST; for example, the program in Figure 4.1a has the CFG shown in Figure 4.1b. However, rather than just using CFGs to represent concrete programs, I used them here to represent the abstract semantics of a PL construct. For example, Figure 4.1c shows the *semantic* CFGs (SemCFG) for any *arbitrary* conditional, one of which is the concrete CFG in Figure 4.1b, with holes (represented by ?) for whatever code, if any, might execute for the then and else branches. All of the (two) paths through such SemCFGs represent all of the valid semantics a learner could know about conditionals.

With these concepts alone, I could only represent individual PL constructs. Returning to my

idea of representing compositions of pairs of semantics, I created permutations of SemCFGs to represent prevalent compositions that learners might misunderstand. To make a composition, fill the holes in a SemCFG with other valid SemCFGs for that hole:

$$SemCFG_1(hole_1, hole_2, ...) \rightarrow SemCFG_1(SemCFG_2(hole_1, hole_2, ...), SemCFG_3(hole_1, hole_2, ...))$$

Thus, these compositions fill one or more of the the unspecified ? subgraphs with other SemCFGs. For example, Figure 4.1d shows a *composite* SemCFG that represents a nested conditional with no outer else branch. All of the (three) paths through this composite SemCFG represent all of the valid semantics a learner could know about conditionals nested one level deep.

By defining a collection of composite SemCFGs to detect many possible misconceptions of a PL's semantics, I built a scoring model that covers all possible paths through all of the SemCFGs I chose to include in my test, giving a 1 or 0 for each path that a learner can or cannot mentally simulate correctly. I defined a notation for these SemCFG paths as nested lists of SemCFG paths, since interpreter execution paths are just node-edge combinations in the graph. For example, for the nested SemCFGs in Figure 4.1d, the three paths in this notation are: 1) *if-true{if-false}*, 2) *if-true{if-true}*, and 3) *if-false*. My proposed scoring model has a 0 or a 1 for each of those paths. I defined correctness as executing a complete valid path for each node in the SemCFG.

To account for data flow semantics, I further extended this model by adding a limited set of *data flow path elements*, which are data flow patterns representing variable assignments and references. These can be included in the unspecified subgraphs of SemCFGs (the ?'s in Figure 4.1c & 4.1d) to help model interactions between control flow and program state. To express data flow, I included in the SemCFG path notation the *scope* that was accessed; for example, *varset-local* represents setting a value in local scope, *varset-global* a value in global scope, and likewise for *varget-local* and *varget-global*. To express more complicated data flow, I encoded which scopes were available for a variable resolution, for example *varset-local-shadowed* for when a local variable was set but it also shadowed a global variable (this represents the nodes available for data flow at that point) (see Figure 4.2c for code whose SemCFG has a varset-local-shadowed and later Figure 4.3). While

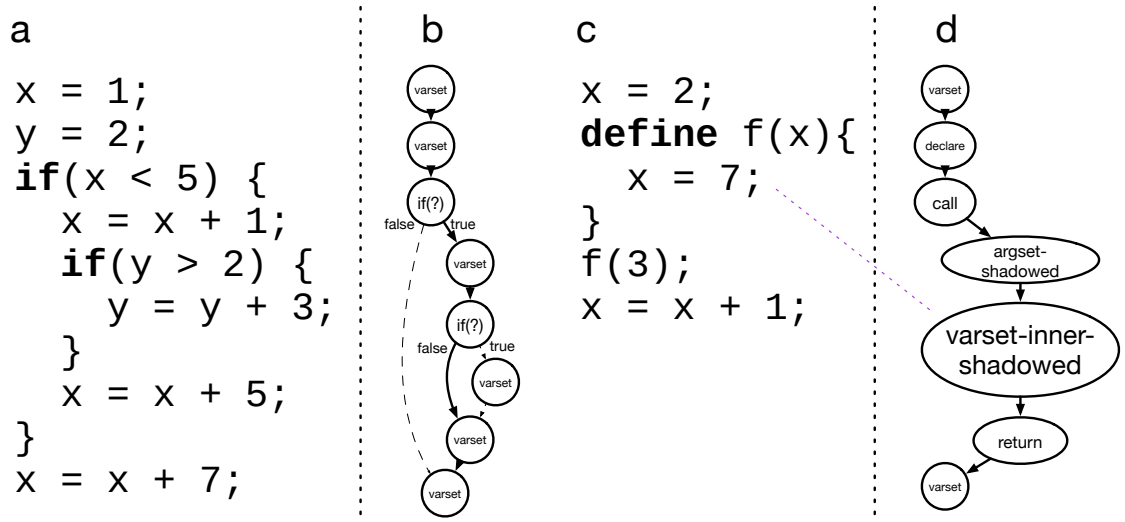


Figure 4.2: (a) An example item (item 28) and (b) the SemCFG path it attempts to assess. (c) A code example with (d) SemCFG with data flow varset-inner-shadowed.

these do not represent *all* data flow scenarios in a PL, I focused on important cases for common PL semantics where novice learners might benefit from formative feedback if they got it wrong (as that is the test's intended use, per Kane's framework).

Combining all of these concepts, my theoretical scoring model represents whether a learner can accurately trace each path through a set of SemCFGs, in order to detect many possible misconceptions of PL semantics, at a fine-grained level of detail. This granularity is critical to identifying specific misconceptions of both individual PL constructs and compositions of PL constructions.

4.2.3 Item Design

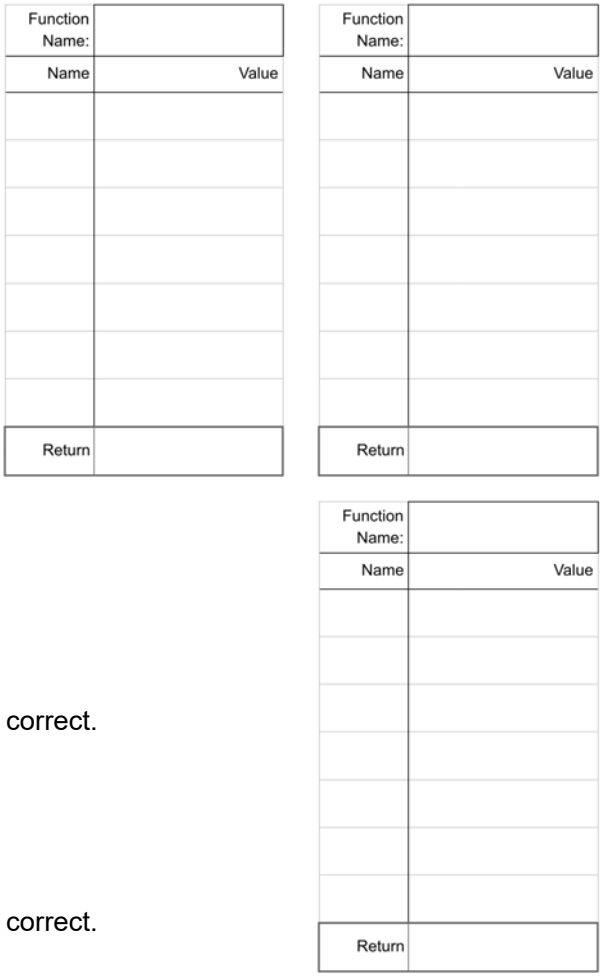
The scoring model in the previous section represents the skills to test. To actually observe these skills and score them, I needed items. Kane's framework says I should argue how my item structure matches the scoring model. Therefore, here I extend my scoring argument by discussing how to design items compatible with the scoring model that minimize confounding factors not accounted for in the scoring model. Figure 4.3 shows an example on the next page.

For the program part of my item design, most of the items followed the program structure in

26

Execute the code and enter what values are in the variables when the program finishes. Fill in the answer boxes on this page (at the bottom) - you should copy the values from your memory table sheet.

```
var x = 3;
var y = 6;
function f(x) {
  x = 7;
  return 2;
}
y = f(5);
x = x + 1;
```



Your Answers:

x is _____

☐ I made a random guess
☐ I made an educated / informed guess.
☐ I am reasonably confident my answer is correct.

y is _____

☐ I made a random guess
☐ I made an educated / informed guess.
☐ I am reasonably confident my answer is correct.

Figure 4.3: Item 21 on the tracing assessment, with tracing scaffolding tables on right and answer spaces on bottom.

Figure 4.2a. Figure 4.2a shows an example of code that assesses the true-then-false path (*if-true{if-false}*) of a nested conditional (represented by the SemCFG in Figure 4.2b). The program's `x` and `y` variables serve three roles. First, their values ensure a particular control flow path is followed; for example in Figure 4.2a, `x=1` ensures the first `if(x<5)` is true, and the y value of 2 ensures the `if(y>2)` is not true, which makes the code follow *if-true{if-false}* instead of *if-true{if-true}*. Second, the variables are the basis of the two prompts for each item; for example, "*x is ...*" and "*y is ...*" in Figure 4.3. Third, by modifying one or both variables before and after each branch in the program, the item ensures that every unique control flow path the learner *might* follow while mentally simulating the program (correct or incorrect) would result in different values for `x` and `y`, to enable unambiguous interpretations of the item response. This item design helps ensure that if the learner knows the values in the variables at the end of execution, one can infer the learner knows all the semantics required to compute those values.

To generate most items for the types of *SemCFG* paths I defined in the scoring model, I followed a program template that supported at most two PL constructs (e.g., a single conditional, a conditional inside of a loop, a function call inside of a loop, etc.):

```
varset path { varset path { varset } varset } varset
```

where each `varset` was the assignment of `x` and/or `y` and each `path` represented the branch of a control structure (e.g. *if-true*, *while-false*). The brackets `{}`'s represent potential nesting. For example, in the nested conditional example in Figure 4.2a, I instantiated the template as `update-x update-y if-true { update-x if-false { update-y } update-x } update-x`. I then translated these instantiated templates into the program like the one above, using inequality expressions to ensure the desired branch, and assignments to ensure non-overlapping values of `x` and `y` for all possible incorrect executions of the program (for example, see Figure 4.2a).

The template above did not work for all of the semantics I wanted to assess (e.g., local and global variables, shadowed variables, function calls and returns), but I generated similar templates for such programs that similarly mitigated confounds. For example, the function call and return items used global variables to isolate knowledge of scoping, function arguments, and return values,

such as the item in Figure 4.3.

While the item design above avoids some confounds, it also introduced some. In particular, it introduced concrete program elements such as literal values, operators, and expressions that may result in *slips* and *guesses*, which confound interpretation of item responses (these terms come from models used in educational assessment). *Slips* occur when some aspect of an item causes a learner's answer to be incorrect, even when the learner can trace the item (for example, misreading an operator, or overlooking a return statement). *Guesses* occur when one infers incorrectly that the learner knows when they do not (for example, making an arithmetic error that inadvertently led to a correct answer).

To mitigate the influence of confounds that cause slips and guesses, I developed the following principles for what program literals and operators to choose to fill in the item templates:

1. **Arithmetic:** For any arithmetic, I preferred small numbers and simple operations to maximize the likelihood people know the operation and minimize working memory errors. While arithmetic errors *might* indicate some misunderstanding when assessing the + operator, arithmetic errors confound measuring more advanced constructs. I also avoided negative numbers, as they likely lead to more arithmetic mistakes.

2. **Perception:** Tracing requires accurately perceiving syntax (e.g., not mistaking `x` for `y`). To address this, I ensured that small errors in variable look up still led to correct performance for conditionals. For example, in the program `x=4, y=3, if(x<5){ }`, the control flow will always do the inner body of the `if`, even if the learner accidentally looks up `y`.

3. **Idioms:** Some patterns in programs are so common, they allow learners to "short circuit" program tracing. To avoid this, I avoided idioms when possible.

4. **Metacognition:** Learners can guess or slip based on cues in an item. For example, variable names like *counter* can give clues to behavior (or be distractors that may cause a slip); or a learner may doubt their answer if no variables change. To avoid this, I used single-letter

variable names, and for larger nested items I ensured one or more variables modified from their initial value.

5. **Guessing:** To motivate honest responses, I explained in the test's cover page that guessing makes it harder to infer what learners know.

4.2.4 Test Design

Given the scoring model and item design, this section now discusses my proposed test design for a practical subset of JavaScript semantics and a practical set of compositions of JavaScript language constructs. Kane's framework says to describe the sample of items, and describe ways to lower score variation to improve *generalization* (and potentially mitigate unintended effects like harming self-efficacy). The argument below therefore constitutes my *generalization* argument. I do not make an extrapolation argument, although I describe steps for one in this chapter's discussion section.

Table 4.1 lists all of the items in the test, corresponding to a set of SemCFGs. The test covers variable assignments, arithmetic operators, inequalities, conditionals, loops, and functions, and prevalent compositions of these constructs. I generated all items using the procedures and guidelines described in Section 4.2.3.

In addition to these items, the test attempts to remove additional confounds by teaching a tracing strategy and scaffolding tracing on each item. The test giver teaches the tracing strategy described in [Xie et al., 2018], using a written script for proctoring the test and not giving personalized feedback on mistakes learners were making. The script instructs the proctor to show an example of the strategy by writing it in pen, then having the learner try the same example, for three examples.

To support a tracing strategy during the test, the items include memory tables [Xie et al., 2018] on each page to scaffold notional machine state recall and representation (see right of Figure 4.3, two pages back). I placed those tables next to each problem to make them at-hand, to reduce visual working memory effort in going back and forth between where they were in the code and making changes to the memory table. The test giver also provides extra table sheets.

To mitigate confounds of the test format:

- The test has no time limit, reducing time-management confounds; this may also improve fairness for people with cognitive differences and perceptual differences like low-vision.

- The test is typeset using a larger mono-spaced font, balancing the trade-off of making character recognition easier and distance needed to move perception to keep track of location and moving between the code and the memory tables.

- To mitigate confounding effects of self-efficacy, the test items are ordered from least to most difficult, to avoid initial shock and dismay; the test cover page also states "Whatever you know right now about programming, you can learn more with more practice. Your performance on these questions has NO RELATIONSHIP with your ability to learn programming." and "The purpose of this assessment is to help you focus your learning on parts you are less strong with at the moment. Please give your best effort, work carefully, and use the explicit tracing strategy we will show you on the next page."

Given the item design for resisting guesses and slips and the mitigation strategies, I believed it analytically plausible that the test scores will generalize within the scope of items on the test, but future work needs to empirically evaluate this. I do not present an argument that the test scores extrapolate, but describe steps for one in this chapter's discussion section.

4.2.5 Scoring and formative use of the assessment

As summarized in Section 2.2.2, Kane's framework says I need an *argument for formative use*, describing how the assessment can target weak skills to improve learning, ideally without indirectly harming other outcomes like self-efficacy. Given my assessment design based on a fine-grained scoring model, in this section I describe 1) how to score the assessment for a learner, and 2) how to use the score to target weak skills with remedial instruction and practice. (This constitutes the *use* argument).

After the learner takes the assessment, one can use each item's overall correctness to assign a *knows* or *does not know* to the SemCFG that item was constructed to assess. If the learner correctly

#	Item template and *added SemCFG information*	
1	x=c1>c2	*booleanExpressionFalse*
2	x=c1>c2	*booleanExpressionTrue*
3	x=c1; x=c2	
4	x=c1; y=x-c2	
5	x=c1; y=x+c2	
6	x=c1; y=c2; y=x-y	
7	x=c1; y=x; y=c2	
8	x=c1; y=c2; if(x>c3) { y = y -c4}	*if-false-no-else*
9	x=c1; y=c2; if(x<c3) { y = y +c4}	*if-true-no-else*
10	x=c1; y=c2; if(x>c3) { y = y +c4} else { x=y+c5}	*if-true-has-else*
11	x=c1; y=c2; if(x>c3) { y = y +c4} else { x=y+c5}	*if-false-has-else*
12	x=c1; y=c2; if(x>c3) { y = y -c4} x = x +c4	*if-false*
13	x=c1; y=c2; if(x<c3) { y = y +c4} x = x +c5	*if-true*
14	x=c1; y=c2; while(x>c3) {x=x-c4}	*while-true while-true while-false*
15	x=c1; y=c2; while(x>c3) {x=x-c4}	*while-true while-false*
16	x=c1; y=c2; while(x<c3) {x=x-c4}	*while-false*
17	x=c1; y=c2; while(x>c3) {x=x-c4} y = y+c4	*while-false*
18	x=c1; y=c2; while(x>c3) {x=x+c4} y = y+c4	*while-true while-true while-false*
19	x=c1; y=c2; f(z){return **z**+c3} y=f(c3)	
20	x=c1; y=c2; f() {x=c3} y=x+c4; f();	
21	x=c1; y=c2; f(x){**x=**c3; return c4}; y=f(c5}; x=x+c6	
22	x=c1; y=c2; f(x){**x=**y}; y=y+c3; f(c4}; y=y+c5	
23	x=c1; y=c2; f(z){y=**z**} f(c3); x=x+c4	
24	x=c1; y=c2; f(x){**x=x**+c3; return g()}; g(){return x}; y=f(c4); x=x+c5	
25	x=c1; y=c2; f(z){y=g(**z**); return **z**+c3} g(k){ return **k**+c4}; x=f(c5}	
26	x=c1; y=c2; if(x<c3){y=y+c4; if(x<c5){y=y+c4} x=x+c5;} x=x+c6	*if-true { if-true }*
27	x=c1; y=c2; if(x<c3){y=y+c3; if(x>c4){y=y+c5} x=y+c4;} x=x+c4	*if-false*
28	x=c1; y=c2; if(x<c3){x=x+c4; if(x<c2){y=y+c3} x=x+c5;} x=x+c1	*if-true { if-false }*
29	x=c1; y=c2; f(z){ if (**z** < c3){ return **z**;} return f(**z**-c3);} x=f(c4);	*recursion with 2 recursive calls*
30	x=c1; y=c2; f(z){ if (**z** > c3){ f(**z**-c3);} return **z**-c3);} x=f(c4);	*non-idiomatic recursion with 2 recursive calls*

Table 4.1: Test items, including the semantic paths they covered. The SemCFG for item 21 is *varset-global varset-global declare varset-global(call{argset-shadow varset-inner-shadow return{const}) varset-global*. For readability the item templates use color for data flow paths local and local-shadowed. I also list parts of the SemCFG in italics when ambiguous, e.g. for item 15 showing the loop executed once with *while-true while-false*. $c1, c2, c3, ...$ are integers chosen to fit the item design principles (Section 4.2.3).

gives the values of variables asked for by the item, then assign *knows* for the item's SemCFG in the scoring model. If the learner gives any incorrect value, assign *does not know* for that item's SemCFG in the scoring model.

To use the score to target weak skills with remedial instruction and practice, give learners an instructional design specific to the SemCFG, with instruction, a worked example of how to trace code that conforms to the SemCFG, and some practice. For example, for item 15 in Table 4.1, one might give a worked example for tracing a while loop that only executes once and updates a variable in the body of the loop, give conceptual instruction for while loops such as "while loops can execute zero, one, or many times, depending on the condition", and give additional practice problems.

The test might be used in many ways and contexts. Instructors might use the assessment or subsets of it in classroom contexts or in lab, to decide what remedial instruction to include in the next class session. The items might also be useful on exams, as they have more granularity than typical exam questions [Luxton-Reilly and Petersen, 2017]. Within a specific classroom context, one can also envision instructors giving subsets of the assessment throughout a course, corresponding to material covered by that time, then using scores to recommend remedial instruction and practice for those items only (such as a lookup table for each test item with what practice to do if incorrect). Instructors might also give parts or the entire assessment as part of reviewing for midterm or final exams. Lastly, learners might use the test themselves to diagnose their own knowledge, then lookup relevant practice.

4.3 Evaluation

The previous section detailed a new approach to designing, generating, and scoring items for a granular assessment for tracing skills. In this section, I evaluate this design idea in three ways:

- **RQ1** What are the item validity characteristics for scoring the item designs, such as slip and guess rates?

- **RQ2** How well can the assessment identify weak or incorrect tracing skills?

- **RQ3** When guided by the assessment, what are the immediate effects of targeted formative instruction and practice on tracing performance, mindset, and self-efficacy?

4.3.1 Method

Best practices in educational assessment research for evaluating item validity (RQ1) typically involve using a (very) large sample to statistically analyze the assessment's psychometric properties. This approach had three issues for my assessment. First, there was no prior work on the appropriateness of underlying assumptions of the statistical methods for the domain of program tracing. Second, it would give little insight on how or why items are problematic (just *that* they were), which makes refining the assessment difficult. Third, by my estimates, the large sample size required for the granular scoring model is infeasible because of the high dimensionality of the scoring model. In consulting with educational measurement collaborators, a simulation study for similar number of items per dimension recommended sample size of 5,000 due to the number of latent parameters that cannot be directly observed in the data.[2]

Due to these issues, I instead chose a mixed-methods study design, which would provide more qualitative insight for improving the assessment, at the expense of quantitative certainty. My study design involved a formative pre-test using the test design, a learning intervention based on the outcome of that pre-test, and a post-test to measure changes in performance (supporting RQ2 and RQ3). Then, I qualitatively observed slips and guesses to inform RQ1.

4.3.1.1 Procedure

I conducted each session one on one. Each participant's session was done in a small office with the door closed, about 8 feet by 14 feet, with fluorescent and natural lighting, and a Molekule air purifier running at a low noise level. The participant used a 5 foot by 3 foot desk placed facing a blank wall. After welcoming the participant, I gave them one version of the test, observing how they solved the items in detail, monitoring for slips and guesses. I observed the participant from the

[2]Using a rough extrapolation from simulations from the table on page 56 in Galeshi's thesis [Galeshi, 2012].

side and slightly back, and took care to not move quickly or noisily to minimize disruptions. Based on the performance on the test, I selected a subset of a JavaScript programming language tutor (PLTutor [Nelson et al., 2017]), which taught semantics the test had identified as weak. After the intervention, I gave the participants a "form B" of the test (which only modified literals) to measure improvements in tracing performance.

For the learning intervention, to select which part of the tutor to have a participant complete, I scored the pre-test to identify incorrect items. Because each item isolated specific JavaScript semantics that the test indicated were weakly understood, I randomly chose one of these weakly understood semantics, and provided the subset of the tutor that explicitly taught those semantics. Additionally, because most of the participants had a strong understanding of the most basic CS1 concepts (e.g., variable assignments, conditionals without else branches), I focused only on the more advanced semantics in the test (function returns and scoping questions and recursion). The PLTutor instruction generally showed three uses of a specific language construct, one discussed as a learner stepped through its execution, and two that required learners to predict the behavior of the construct. Before starting instruction, I had participants trace the first example with pen and blank paper, redoing the item they had gotten wrong on the pre-test (used as *lesson-item* correctness data later).

Additionally, before and after each test, I measured changes in programming mindset [Scott and Ghinea, 2014], fixed vs. growth mindset [Blackwell et al., 2007], and programming self-efficacy [Tsai et al., 2019] (we chose this self-efficacy instrument as it avoids mentioning classrooms and is less writing focused).

During the post-test, I followed a think-aloud protocol, giving participants think-aloud warm-up questions as described in [Ericsson and Simon, 1993], and then prompting them to think-aloud. After the post-test, I gave a survey that measured demographics and confounding factors. These included Likert-like (scale 1-5) agreement questions for *"The tutorial between the two assessments helped me learn something new or fixed a misunderstanding I had."*.

After the survey, I had participants reattempt items for the participant had different pre-test and post-test correctness, and asked them clarifying questions. If the participant had a potential slip or

guess based on the researcher's observation, I gave that item, otherwise the pre-test or post-test item that was incorrectly done. I call these the *post-check* data.

This procedure yielded item response data for each participant that included the item number, the test variant, a binary correctness outcome for the two prompts on each item, *post-check* correctness (if it was given), and learner-reported certainty (random guess, educated/informed guess, and reasonably confident (discussed in Section 4.2.3). I had *lesson-check* data for the pre-lesson item with correctness and item number only. The surveys gave all of the measures of mindset and self-efficacy, the post-survey learning question, participant item copies with marks (like their memory tables), and hand-written researcher notes on each item on participant response process and potential slips and guesses.

4.3.1.2 Participants

We recruited at one large public university from Java-based CS1 and CS2 classes, as well as an R-based introduction to data science course that had students who might also be learning JavaScript in other classes. Sessions occurred in March 2019.

We recruited 31 participants (19 identified as men, 12 as women; 19 undergraduates in year 1, 6 in year 2 , 2 in year 3, and 4 in year 4, ages 18-26). This sample is larger than typical qualitative validation studies (e.g. 7 in the FCS1 validation [Tew and Guzdial, 2010]). Participants had widely varied prior learning experiences, with self-reported hours of learning programming or programming ranging from 10 participants with $\leqslant$50, 14 with $\leqslant$100, 26 with $\leqslant$200, and 5 with $>$200. Prior Javascript experience varied, with 8 reporting CSS/HTML, and 3 reporting Javascript when asked on the recruiting survey.

4.3.2 Results

4.3.2.1 RQ1 Item validity

To investigate these item validity characteristics, I analyzed slips and guesses (as defined in Section 4.2.3 and further below) then estimated the slip and guess probabilities for each item. Both are

key to understanding the extent to which item responses were indicative of knowledge or other confounding factors.

To identify guesses and slips, I triangulated several sources of data:

1. Observations of test performance on the pre- and post-test and post-check items, both in-person during the study and reviewing participant test markings.

2. Qualitative observation of how each participant answered the items. This included what marks a learner made on paper and where they looked, and their think-aloud during the session (the audio was also recorded for review after the session).

3. Qualitative observations of the the learner engaging in the personalized lesson.

4. A semi-structured qualitative interview about their varying test performance, done after post-check questions when time allowed.

To code guesses and slips, I used a set of guidelines for each. I coded answers as a slip when the learner got the item wrong but got similar items correct across the pre and post test sessions (especially without any instruction on the item). Another strong signal for a slip was when I observed small local deviations from the correct response process, like writing down the wrong literal value or evaluating a condition incorrectly when they generally did them correctly. For example, when measuring the SemCFG `if-true{varset}` with the item `x=8; if(x<10){x=x+1;}`, miscalculating `x<10` as false would lead to a slip, because < is not in the SemCFG I designed the item to measure. If I were testing SemCFG `if-true(<){varset}`, it would be an incorrect answer. In general slips were easier to code because the test taker would demonstrate correct performance and the slip would be an exception, often with a particular local mistake.

Identifying guesses was more difficult, because a guess happens in the *absence* of a correct tracing process or conflicting attempts to trace. Thus, I made more substantial guidelines; if any applied I coded the answer as a guess:

1. The learner got most related items wrong on that copy of the test, or on their other test

2. The learner marked the "random guess" for answer certainty for the item and showed inconsistent semantics in their tracing strategy marks (in the tracing tables or otherwise)

3. The learner said in think-aloud that they were guessing or used incorrect logic that did not align with correct semantics, I marked it as a guess.

4. The learner gave an answer without showing evidence in memory tables or other marks on paper or in think-aloud, when they otherwise routinely showed such evidence.

For example, a learner traces code using the memory tables confidently and quickly for several questions, until they reach a new question where they pause, make several false starts and write conflicting traces on the paper, then sighs and choose one answer from one trace attempt and the other answer from the other trace attempt. When those answers were correct, I coded that as a guess. I still may have missed guesses, so the coded guesses probably represent a lower-bound.

To estimate the reliability of the coding procedures above, I generated a random sample of 22% of the participants (excluding 2 that lacked think-aloud recordings), and had a second rater[3] redundantly code. The 1st author went through and blacked out notes about slips or guesses, so the second rater only had participant test copies and think aloud, post-check and lesson performance on practice questions, along with the guidelines above. Cohen's kappa was .434, which is "moderate" [Viera and Garrett, 2005].[4]

In order to estimate the slip probability for each item, I used the following method for estimating a population proportion using a binomial distribution. Assuming the slip probability, *P(item incorrect | can trace the item)*, only varies per item, the observed count of slips in the sample follows a binomial distribution with n = number who can trace the item, and k = number of slips. The

[3] Andrew Hu, an undergraduate research collaborator

[4] Note that I had more information when making my qualitative coding than the second rater, as I was physically present during the testing sessions, in particular timing information and variations in facial gestures. For this study I did not video-tape the participants due to Institutional Review Board Human Subjects complications and difficulty anonymizing such recordings, and I also thought recordings would be highly unusual for participants and make their response process less externally valid. Since the ultimate goal of the study was to evaluate how the test might be useful in actual use, I tried to minimize those issues.

item #	slip probability [conf. int.]	guess probability [conf. int.]
1	0.039 [0, 0.1]	0.020 [0, 0.08]
2	0.044 [0, 0.11]	0.017 [0, 0.07]
3	0.008 [0, 0.03]	NA
4	0.008 [0, 0.03]	NA
5	0.025 [0, 0.06]	0.250 [0, 0.77]
6	0.008 [0, 0.03]	0.250 [0, 0.77]
7	0.008 [0, 0.03]	0.250 [0, 0.77]
8	0.008 [0, 0.03]	NA
9	0.008 [0, 0.03]	NA
10	0.042 [0, 0.09]	0.100 [0, 0.36]
11	0.008 [0, 0.03]	0.167 [0, 0.57]
12	0.008 [0, 0.03]	NA
13	0.008 [0, 0.03]	0.250 [0, 0.77]
14	0.205 [0.11, 0.31]	0.036 [0, 0.13]
15	0.040 [0, 0.09]	0.167 [0, 0.57]
16	0.026 [0, 0.07]	0.071 [0, 0.26]
17	0.008 [0, 0.03]	NA
18	0.042 [0, 0.09]	0.083 [0, 0.31]
19	0.025 [0, 0.07]	0.214 [0, 0.5]
20	0.230 [0.1, 0.36]	0.014 [0, 0.05]
21	0.057 [0, 0.12]	0.023 [0, 0.09]
22	0.036 [0, 0.09]	0.104 [0.01, 0.22]
23	0.012 [0, 0.05]	0.065 [0, 0.16]
24	0.028 [0, 0.11]	0.011 [0, 0.04]
25	0.048 [0, 0.12]	0.015 [0, 0.06]
26	0.137 [0.06, 0.22]	0.050 [0, 0.19]
27	0.062 [0.01, 0.13]	0.045 [0, 0.17]
28	0.073 [0.02, 0.14]	0.083 [0, 0.31]
29	0.034 [0, 0.09]	0.024 [0, 0.09]
30	0.026 [0, 0.1]	0.033 [0, 0.08]

Table 4.2: Results of estimating slip and guess probabilities for each item, with 95% high posterior density intervals (similar to confidence intervals). “NA” denotes that no guesses were observed for that given item.

sample estimators are thus, for each item, $\hat{k}$ = number of slips on that item from the qualitative coding, and $\hat{n}$ = number of observed correct on that item + the number of slips; by using these together with a statistical inference techniques for a binomial distribution, I estimated the probability for each item as $\hat{p} = \hat{k}/\hat{n}$.

Similarly, I also estimated the per-item guess probability, *P (item correct | cannot trace the item)*, with $\hat{k}$ = number of guesses on that item from the qualitative coding, and $\hat{n}$ = number of observed incorrect on that item + the number of guesses.

Table 4.2 reports the slip and guess estimates. Because the study's dataset did not meet inference conditions for frequentist methods ($n\hat{p} \geqslant 10$ and $n(1-\hat{p}) \geqslant 10$), I used Bayesian statistics which require no conditions (except assuming a uniform prior, recommend in related educational measurement work [Zhan et al., 2019]). Table 4.2 shows point estimates and high-posterior density intervals, a Bayesian version of confidence intervals, for all slip and guess rates.

The test's items exhibited excellent measurement properties overall, although the certainty about these estimates varied. The scoring model's parameters align with a Cognitive Diagnostic Model (CDM) [de la Torre and Minchen, 2014, Sessoms and Henson, 2018, De La Torre, 2011]; for these models, .05-.15 for slips and guesses are considered "good" because they can produce good estimation of the scoring model [De La Torre et al., 2010]. However, the uncertainty varied because the study population's tracing ability varied; for example, if only one person could not do an item, I effectively had a sample size of one to estimate the guess rate for that item (e.g. item #7 in table 4.2). Therefore, for items many people knew, I learned much about slip rate but little for guess rate.

4.3.2.2 RQ2 Identifying weak knowledge

To identify how well the assessment identified weak knowledge, I measured the proportion of learners who appeared to have weak knowledge when starting the lesson, using 1) a direct measure of weak knowledge, and 2) a secondary measure to triangulate the first.

The direct measure was an estimate of the likelihood that a learner can benefit from a random lesson selected based on their pre-test performance. I used *lesson-item* correctness data: after the pre-test, learners redid the item they got incorrect as the first part of their lesson; when incorrect

again, I interpreted that as evidence the pre-test had correctly identified weak knowledge, because the learner had twice demonstrated their weakness. I estimated the probability the lesson chosen by the pre-test targeted an item the learner could not do, using the proportion that got that item incorrect. Using a binomial model to estimate this probability, the resulting estimate is .734 CI[.582, .878] for the probability that a learner that randomly gets instruction targeted by the test may benefit.

In order to gain more confidence in this estimate, I checked if those that got the *lesson-item* incorrect more frequently self-reported learning, compared to those that got it correct. I used *self-reported-learned* from the post-test survey, a 1 to 5 likert question for *"The tutorial between the two assessments helped me learn something new or fixed a misunderstanding I had."*. Of the 24 who got *lesson-item* wrong (targeted weak), 10 reported strongly agree and 11 agreed (over 90%), providing secondary evidence from participant self-reports that they learned from the lesson. For the 8 who got *lesson-item* right (did not target weak), 1 reported strongly agree and 5 agreed (75%), and from interviews they mentioned the notional machine detail of PLTutor, not semantics (for example, one person said they learned "nothing in particular" during tutorial but it "helped to think what computer thinks with step by step process").

These results suggest the assessment identified weak knowledge, as the estimate of probability of potential benefit was .734 CI[.582, .878], sampled from a random selection of the topics identified by the scoring model.

4.3.2.3 RQ3 Impact of targeted instruction

Given evidence that the assessment could identify weak knowledge, how well did this identification of knowledge translate to improved learning?

I wanted to measure learning, by counting people that improved from incorrect on the pre-test to correct on the post-test. However, that has a flaw: because people sometimes slipped on the pre-test, that would mistakenly count that as learning, when their tracing ability did not actually change.

To mitigate that bias, I used the qualitative coding of slips and guesses to "correct" the raw item response data. I created a new data set, *adjusted-response-data*, with the researcher inferred item correctness, by changing each wrong item response due to a slip into a correct, and replacing each

correct item response due to a guess into an incorrect.

With that data I estimated *P(learn | incorrect on pretest and in transfer-group)*, grouping items by categories for whether the participant's lesson taught related knowledge. I had four groups: *near-transfer* items that covered the exact same path as practice given in the lesson, *far-transfer* items that shared path components with at most one additional level of nesting, *farthest transfer* items that included paths whose rule was stated in natural language in the lesson but not practiced, and *disjoint* for items with no overlap with the lesson.

I estimated *P(learn | incorrect on pretest and in transfer-group)* using a binomial proportion, with k = items in *transfer-group* incorrect on pre-test but correct on post-test, and n = items in *transfer-group* incorrect on pre-test. For the near transfer as the *transfer-group*, I estimated the probability as .775 CI[.625, .915] using k=22 of n=28 items (with zero to one potential item for learning per participant). For the far transfer, I estimated .716 learned CI[.556, .868] using k=21 of n=29 items (with 0 to 3 per participant). For the farthest transfer, I estimated .431 learned CI[.233, .633] using k=9 of n=21 items (with 0 to 3 per participant). For the disjoint items not covered in the lesson, I estimated .252 learned CI[.165, .343] using k=22 of n=88 (with 0 to 12 per participant).

To contextualize these changes, I compared these changes from targeted formative assessment versus just teaching broad topics, which I operationalized by "teaching" function scope in general, by having all lessons related to scoping contain a full verbal description of function scoping rules; targeted instruction with practice compared favorably to this. I operationalized this by estimating then comparing *P(learn)* for farthest transfer versus near transfer items, for the function scoping lesson participants (lessons for items 20 through 25), using the same method as the prior paragraph. The estimated probability for learning for near transfer was .825 CI[.662, .967] using k=16 and n=19. The estimated probability for farthest transfer was smaller, .431 CI[.233, .633] using k=9 and n=21. Participants learned much better on near transfer, compared to far transfer; this change was significant as the 95% intervals do not overlap. Thus, learning with targeted practice seemed to outperform general instruction. This result depends on the particulars of this study's "untargeted" and targeted instructional designs and may vary with other designs.

These changes suggest formative assessment and instruction led to learning as well as some

over-generalization of lessons. Learning from the assessment appears feasible, although it may vary based on lesson and quality of the instruction, and the results here may not generalize to other populations; I gave this Javascript test to students near the end of Java-based CS1 and CS2, with at least 23 of 31 with no Javascript experience.

4.3.2.4 Effects on mindset and self-efficacy

Independent of item performance, at the participant level I evaluated impacts of use for formative assessment on programming aptitude mindset [Scott and Ghinea, 2014], programming self-efficacy [Tsai et al., 2019], and growth vs. fixed mindset [Blackwell et al., 2007], which I operationalized as differences between between before pre-test survey and after post-test. There were very small improvements that were not statistically significant.

Programming aptitude mindset (from 1 for most growth i.e. "I can change my aptitude for programming", to 6 for most fixed i.e. "I cannot change ...") started at mean=1.92, standard deviation(sd)=.67 and became more growth after post-test, mean=1.69 sd=.75. The per individual change ranged from -1.33 to 1, mean=-.24 sd=.55.

Programming self-efficacy (from 3 least to 15 most) started at mean=10.74, sd=1.87 and improved after post-test to mean=11.31, sd=1.92. The per individual change ranged from -1.17 to 3.42, mean=.56 sd=1.

Growth vs. fixed mindset (from 1 for most growth to 6 for most fixed) started at mean=2.74 sd=1.09 and improved after post-test to mean=2.17 sd=1.15. The per individual change ranged from -2 to 1, mean=-.3 sd=.68.

Interestingly, programming self-efficacy dipped very slightly after the first test, to mean=10.59 sd=1.72 measured before the learning intervention, then rebounded after the intervention to mean=11.12 sd=1.93. The other measures showed a steady but small increase in the desirable direction (more growth mindset, more self-efficacy) after each measurement, none statistically significant.

In summary, mindset and self-efficacy went up on average, but the differences were not significant and may have been within measurement error. The instruments I used did not have test-retest

reliability for any further analysis (nor were instruments with this available to my knowledge). This study did not suggest large negative effects on self-efficacy or mindset, for the assessment followed by instruction guided by the assessment.

However, I halted the study for one participant (P8) after she seemed visibly distressed during the pretest, out of an abundance of caution; this may indicate a fat-tailed non-normal distribution for effects of giving the test in a context where the learner hasn't received instruction and practice on the test's programming language before. I halted the study for one learner when she went over 50 minutes on the first test and was visibly distressed (she was stuck repeating a question that she did not know). In an interview afterwords she reported going to a high school in China which emphasized extremely fixed mindset: "effort was for failures." The assessment was on Javascript but the class I drew participants from was for Java, and she had not explicitly learned the programming language on the test previously. When we discussed this she reported feeling better. Her pre-test survey also did not below average values on self-efficacy, and showed moderate growth mindset for both programming aptitude mindset, and growth vs. fixed mindset.

4.4 Limitations

This evaluation was confounded by many factors. I only evaluated effectiveness for one lesson, whereas in practice one would give lessons for all weak knowledge areas identified by the test. Many of the results were likely impacted by learner and environmental factors I measured and did not measure, both those in the model of program tracing in section 4.2.1 and the more complex model presented earlier in that section, and thus may not generalize. Prior knowledge strongly influences effects and most of the participants had little JavaScript experience. The volunteer participants likely had more growth mindset than other populations, and prior mindset and self-efficacy likely mediate learning.

Another limitation is that the assessment ignored the many strategies for comprehending and tracing programs. For example, the post interviews revealed that, when uncertain, people sometimes came to single interpretation and corrected past questions, but sometimes did not fix past answers even when they realized they were wrong. This layering of different comprehension strategies threat-

ens item characteristics generalizability and inference from the test [Borsboom and Mellenbergh, 2007]. Another study with think-aloud on the first test could clarify the validity argument for this test and might suggest ways to change the item or test design to reduce this variation.

The test also assumed that the skills assessed do not change during the course of the test. Of course, this is rarely true (I observed it in 6 participants). Future work should measure to what extent this leads to incorrect inferences in the scoring model.

The use of think aloud was also limiting. Think aloud is known to change what people are thinking when the information being shared is not verbal or propositional [Ericsson and Simon, 1993], which may change test results. At the same time, in our data, many participants sped up their response times with think-aloud, which suggests that the majority of the problem solving for each item was verbal or propositional. Both concurrent and retrospective think aloud might gather a fuller picture of student problem solving [Ericsson and Simon, 1993, Taylor and Dionne, 2000].

Finally, while the item generation approach works for an imperative language like Javascript, it may not generalize to constraint languages and pure functional languages. I also did not attempt to extend item generation to dataflow within expressions, focusing primarily on control constructs.

4.5 Discussion

The main contributions of this chapter include a novel design of a formative assessment of tracing and evidence that it can effectively target and support improving weak tracing knowledge. The core idea behind the approach is building a bridge between a basic theory of PL knowledge and the varying ways in which understandings of those semantics can be incorrect, for nested combinations. This work is also the first application of Kane's framework in computing education research.

These results, despite the limitations discussed in the previous section, have some immediate implications for assessment in K-12 and higher education computing education. For example, the lack of prior formative assessments of program tracing with validity arguments means that our test design could be used in practice immediately, as it likely provides a more precise (and efficient) measurement of program tracing knowledge than existing assessments. This is especially true for instructional design approaches that explicitly teach program tracing (e.g., [Nelson et al.,

2017, Xie et al., 2018]). Future work might even explore ways of automating the selection of learning interventions, creating tight learning feedback loops that rapidly accelerate effective learning of programming languages semantics.

I plan to prioritize improving the generalization and extrapolation argument by studying response process via think-alouds for confounds on performance. I should evaluate extrapolation and use claims such as to what degree formative use for test-like tasks supports learners in mental tracing during debugging, broader program comprehension tasks, and program writing. While I hypothesize using the assessment formatively may support that later learning of those more complex tasks, I should study deploying the assessment and gather direct evidence on changes in later learning outcomes.

While there are significant opportunities for improving the assessment and gathering evidence for improving the validity argument further, Kane's conception of validity [Kane, 2013] as the quality of an *argument for effectiveness for some use* should lead the computing education research community to consider more than just accuracy and reliability. As a first example, if our goal is to use an assessment to formatively support learning, even if our assessment had perfect accuracy but negatively impacted self-efficacy or mindset in the process, it might have a net negative effect on long-term learning, and therefore might not be "valid" for that use. As a second example, even if including strategy instruction and scaffolding produced some more slips initially on the test due to unfamiliarity, which is undesirable for traditional notions of validity as accuracy, prior work shows that teaching tracing strategies can help learners [Xie et al., 2018], so including it may still make for a more "valid" assessment for the goal of improving learning. The question is therefore what accuracy is good enough for particular uses of the assessment, and what are the trade-offs and opportunity costs? As a third example, using Kane's framework led me to expand my considerations beyond just accuracy and reliability, to also include self-efficacy, mindset, and other learning outcomes, which have not typically been measured in knowledge assessment studies in CER (e.g., [Tew and Guzdial, 2010, Parker and Guzdial, 2016]).

Kane's framework emphasizes considering extrapolation carefully; computing education research should further investigate assessments that extrapolate well to real-world contexts. For

example, writing code is actually done with IDEs that can run code; why then assess writing and tracing code without IDE features [Piech and Gregg, 2018]? Some past work has tried more authentic IDE-based assessments in labs [Prior, 2003, Jacobson, 2000]; to study extrapolation, work should ultimately compare test performance with performance in job or other real-world settings. As another example, my work in this chapter is guilty of assessing tracing as an abstract skill, rather than situating it in more real-world contexts of program comprehension, testing, debugging, and other authentic activities that occur in programming and software engineering contexts. My only defense of this choice is that the skills required for the items seem so fundamental that people need to learn them robustly to learn later concepts (e.g. [Lister, 2016]); this may not be true, nor is it necessarily true that all in computing can be abstractly learned then practically applied. We should seek assessments that extrapolate to real-world contexts, lest we teach only easy-to-assess tasks and skills.

Beyond improving the formative assessment of tracing, my work may have implications for other forms of assessment of computing knowledge. For example, a key idea in my work was building cognitive models of tracing ability out of a theory of basic PL knowledge. Future work could explore similar approaches for program *writing*, building upon theories that decompose writing into other skills [Xie et al., 2019b, Mead et al., 2006, Luxton-Reilly et al., 2018a].

As computing education researchers develop and improve both formative and summative assessments of computing knowledge, it is important to think broadly about their use. We should not only consider factors like accuracy and reliability, but also a much broader diversity of learning outcomes and how assessments shape learning over time (including equity and intersectionality in the effects of assessments, although my initial evaluation has not done this). My contributions are small example of how to do this; I hope my community will explore many more.

To use the tracing assessments or develop new ones for other programming languages, please send an email to glnelson@uw.edu and I will happily share them with you. You can also recreate the test using the item design principles at the end of Section 4.2.3 and the item listing in Table 4.1.

Chapter 5

DIFFERENTIATED ASSESSMENTS: JOINT ASSESSMENT OF ADVANCED PROGRAMMING SKILLS WITH PREREQUISITE PROGRAM TRACING SKILLS

In this section, I extend my theory of programming language knowledge to include relationships with advanced skills, identifying the problem that assessments of advanced topics aren't precise enough to distinguish between learner issues with an advanced skill versus its prerequisites. Based on that theoretical insight, I co-led a collaboration[1] to 1) establish the problem exists via an empirical qualitative study of existing assessments with respect to prerequisite program tracing skills (see Section 5.4), and 2) create example questions and design guidelines for *differentiated assessments*, designed to distinguish between advanced and prerequisite program tracing skills (see Section 5.5 and Section 5.5). This work was peer-reviewed and published in the ACM Innovation and Technology in Computer Science Education conference [Nelson et al., 2020].

5.1 Introduction

Assessments for advanced courses aim at assessing specific advanced topics, even though they often require prerequisite knowledge or skills. The cause for a learner to fail such assessments may relate to 1) difficulties with the advanced concept or 2) weaknesses with prerequisite concepts that may lead to incorrect application of the advanced concept. For example, suppose the advanced concept involves synchronizing data in a concurrent program, and the prerequisite weakness is scoping (which variables are shared across function calls/class instances). An advanced assessment might

[1] In this chapter I use "I" to refer to work I primarily developed (conceptualizing differentiated assessments and the gap in related work) as part of discussions with Filip Strömbäck during the Koli Calling 2019 doctoral consortium, conference, and pleasant train ride back from the conference. I use "we" to refer to work I co-led with Filip Strömbäck and Ari Korhonen as part of a multi-national collaboration with Marjahan Begum, Ben Blamey, Karen H. Jin, Violetta Lonati, Bonnie MacKellar, and Mattia Monga.

ask a learner to add concurrency controls to a piece of code that reads a shared (perhaps global) variable and adds the value to a local variable. If the learner thinks the shared variable is not shared and doesn't add enough concurrency controls, the learner gets the question wrong due to weakness with scoping, not a misunderstanding of how to add concurrency controls or how concurrency controls work. The learner also may arrive at a working solution even while holding the belief that local variables are shared; that can be a missed opportunity to identify and correct a belief that will cause problems later.

However, assessments for advanced topics are not intended to assess prerequisites explicitly, and may not support diagnosing the root cause. For example, concept inventories for advanced topics seem unlikely to already diagnose problems with prerequisites, for two reasons. First, while concept inventories for advanced topics include distractors for misconceptions, they are designed by definition to cover misconceptions about the advanced concepts, not the prerequisites. Second, most concept inventories are actually never evaluated for how well they can diagnose student thinking or give formative feedback to students; just because they measure knowledge does not imply that measurement is useful for giving feedback [Santiago Roman, 2009, Nelson et al., 2019, Sands et al., 2018, Jorion et al., 2015, Denick et al., 2012]. To our knowledge there has been no analysis in computing education of whether the distractors for a concept inventory might diagnose prerequisites.

Pre-exams are also a potential tool to identify issues with prerequisites at the beginning of an advanced course, with some drawbacks. Administering pre-exams and addressing the identified issues is time consuming, may result in "boring" activities for learners, and learners knowledge may weaken by the time they need to use specific parts of the prerequisites. Moreover, pre-exams may not be enough to guarantee that the prerequisite skills are at the level required to focus properly on the new advanced ones to be learned.

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

In response to these drawbacks, we contribute a new genre of assessment: **differentiated**

assessments, which are advanced topic assessment questions designed to also diagnose relevant prerequisite issues. The key idea is to expand the typical scope we define for an assessment, to cover more of the prerequisites. Unlike giving a separate pre-test at the beginning of a class, differentiated questions would be easier for advanced course instructors to use because they serve both purposes. Ideally, different incorrect answers would diagnose difficulties with the advanced content and difficulties with prerequisites, and also be easy to grade or even generate feedback for automatically. This can be especially helpful for learners with varied backgrounds and weaker prerequisite knowledge; thus, differentiated assessments seem promising for improving equity.

We conducted a design investigation into whether it is possible to design differentiated assessments, and if so, how this might be achieved. We scoped *prerequisite* skills to basic PL knowledge and program tracing skills. For clarity we refer to advanced topics as *course topics*[2] , scoped to algorithms and data structures, advanced object-oriented programming, and concurrency. We also analyzed questions from the Basic Data Structures Inventory (BDSI), a concept inventory for data structures [Porter et al., 2019], which we chose because it is a recent, state of the art assessment with a validity argument supported by empirical studies.

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

RQ1: What prerequisite program tracing skills do advanced CS questions depend on?

RQ2: To what extent can an existing concept inventory for data structures with a validity argument – the BDSI [Porter et al., 2019] – also diagnose difficulties with prerequisite program tracing skills?

RQ3: What are examples of differentiated assessments and principles for designing differentiated assessments for advanced and program tracing skills?

[2]The distinction between prerequisite and advanced skills depends on a given learning context, such as a course. Prerequisite skills are those skills that students are assumed to be familiar with before attending a course, which are different from the new concepts the course aims to teach. For example, as a student progresses through their education, topics that were once new and difficult increasingly turn into prerequisites for other topics in other courses.

5.2 Method

The goal of our work was to explore the issue that assessment for advanced courses often implicitly assess prerequisites, and to design improvements to these assessments to make prerequisites explicitly assessed. For methods, we have adopted the ideas in *educational design research* suggested by McKenney and Reeves [McKenney and Reeves, 2014, McKenney and Reeves, 2012]. According to the authors [McKenney and Reeves, 2014], "educational design research is a genre of research in which the iterative development of solutions to complex educational problems provides the setting for scientific inquiry,"and attempts to solve real-world problems while seeking to discover new knowledge that might be valuable to others facing a similar problem. Since educational design research is not a single method, but rather a way of approaching a problem, there was no well-defined procedure to follow. We follow the method suggested by Reeves [Reeves, 2006], which consists of four phases: *problem analysis*, *solution development*, *iterative refinement*, and *reflection to produce design principles*.

McKenney notes it is essential to involve practitioners in this kind of educational design research in order to properly identify real teaching and learning problems, and to create prototype solutions based on existing principles and prior experience [McKenney and Reeves, 2014]. The participating researchers in the current study are active faculty members and/or active researchers in computing education research with primary responsibility for not only teaching their courses, but also designing the courses and developing course assessments. Together, the researchers represent over 100 years of collective experience in pedagogy and course design. In addition, several of the researchers have had significant responsibility for curriculum design at the program level, in official capacities such as CS program director, service on departmental curriculum committees, and through development of new programs at their universities. As a result, they brought to this task experience in course development at both the course and the program level, as well as years of experience developing course materials and assessments.

Our design process consisted of the four steps that are outlined in Figure 5.1:

Problem Analysis and Data Collection: This phase includes framing the problem (which we

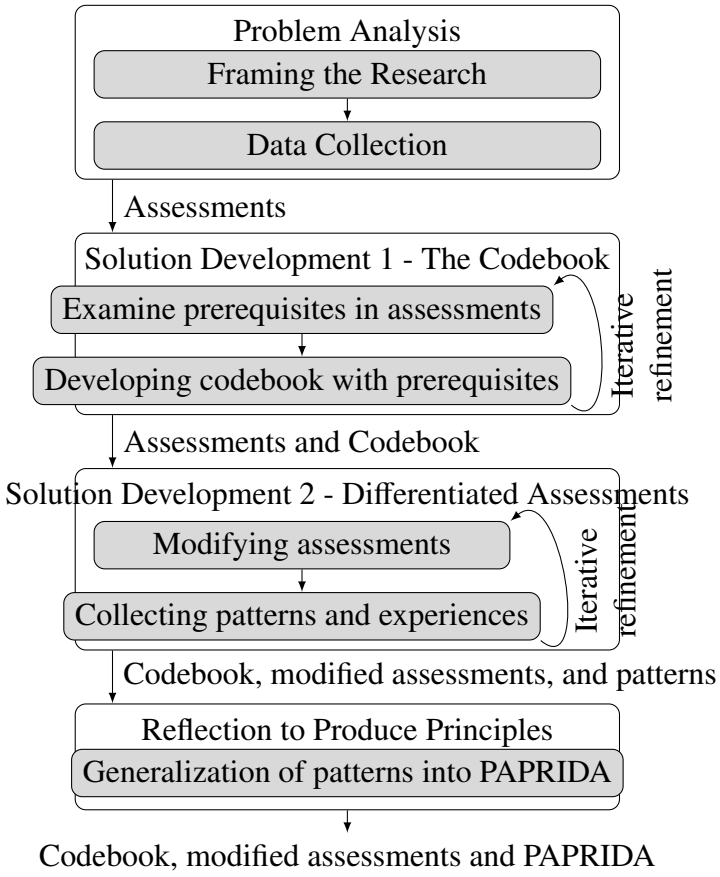


Figure 5.1: Overview of how the educational design research method was applied.

described in the introduction) and collecting data to analyze further.

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

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

phase.

Reflection to Produce Principles - PAPRIDA: Intertwined with the **Solution Development 2** phase where we created differentiated assessments, we also identified and refined PAtterns and PRinciples for Differentiated Assessment (PAPRIDA).

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

5.2.1 Problem Analysis and Data Collection

We analyzed assessments from our undergraduate and masters level courses from a range of institutions in Europe and the United States. We specifically selected assessments covering the topics of concurrency, algorithms and data-structures, advanced object-oriented programming. As previously mentioned, we refer to these as *course topics*, implying that they are what is intended to be covered, compared to *prerequisites*. We limited the scope of assessment material considered to "core" programming-related problems, including those related to pseudo-code. We also selected assessments which were judged to be problematic for students due to lacking prerequisite skills, based on the research team's collective experience in teaching those topics and using those assessments in their courses. Some members had created pretests specifically constructed to assess prerequisite knowledge. In these cases, the pretest questions were also included in our sample of questions, as they may act both as inspiration for how to better assess prerequisite skills in assessments for course topics, and that they may be used as a foundation to build new such questions on top of. We also included the BDSI [Porter et al., 2019] within our sample, to answer RQ2 to see what prerequisites are involved in a state of the art concept inventory. In total, we looked at 11 questions from courses taught by the research team, and all 13 questions from the BDSI. The 11 questions from courses taught by the authors, as well as two questions from the BDSI are available in Appendices M and O. Questions labeled M.x are questions we later made modified versions of to improve differentiation, questions labeled O.x are questions that were analyzed but not modified, and questions labeled B.x are questions from the BDSI.

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

5.2.2 Solution Development 1 - The Codebook

The goal of the prerequisites analysis was to examine which, if any, prerequisites a student needs to understand in order to answer each question properly. We first considered a deductive approach based on a predefined codebook. To find a theoretical or a authoritative source for this codebook, we considered using the ACM 2013 Curriculum Guide [Association for Computing Machinery, 2013], the Core Concepts identified by Goldman et al. [Goldman et al., 2008, Goldman et al., 2010], and the Misconception Catalogue compiled by Sorva [Sorva, 2012] (See Appendix A.1). Each of these potential sources were, however, not deemed suitable for our analysis.

While the ACM 2013 Curriculum Guide is very complete, the main problem for our usage is that a number of topics are described at a high level of abstraction, while we desired a higher level of detail. For example, the question O.7 (available in the Appendix) assesses whether a student has realized that the `return` statement halts execution of the current function in addition to defining what value to return from the function. This skill in particular maps to the ACM Curriculum Guide concept of "SDF/Fundamental Programming Concepts/Functions and parameter passing". This concept does, however, also cover the aforementioned "what to return" and function parameters in addition to halting execution of the current function.

our main issue with Goldman's "Core Concepts" was its scope: only intended to cover introductory concepts. As the specific prerequisite concept for a course varies depending on the level of the course and what is covered in earlier courses, the topics covered by the Core Concepts might not cover all potential prerequisites for our questions. The previously mentioned example with the

return statement illustrates this issue as well: the Core Concepts does not include a concept that involves return, neither the return value nor the act of halting execution.

Finally, the Misconception Catalogue would indeed form a detailed codebook, but due to its high focus on object-oriented programming, we deemed it unsuitable. The example regarding the return statement once again illustrates this: it lists misconceptions regarding the return value, but not regarding the act of halting execution.

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

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

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

When both groups were done with their coding work, the two groups met and discussed their findings, aiming to merge the codes created by the two groups. This corresponds to another design

phase. After this, the two groups independently re-coded all questions using the unified codebook. This resulted in a number of minor revisions to the codes, which were then discussed between the two groups again. This was repeated until all members of the working group were satisfied that the codes covered all prerequisites covered by the collected assessments. Note that due to the iterative refinement of the codebook, some codes were split into two during the process, resulting in codes representing prerequisites that are not assessed by any of the questions. As these still represent valid prerequisites, they were kept in the codebook in spite of the fact that they are not present in any of the questions. The codebook, and our coding of the assessments are presented in Section 5.3.

In order to examine whether the questions in the BDSI had similar issues to the other assessment we collected, we examined the distractors in the BDSI once more after the codebook was finalized. For each question, two researchers independently coded which prerequisite skills, if any, a student picking each distractor might have difficulties with. In order to properly distinguish between prerequisites and course topics, the researchers also noted which course topics a student could have difficulties with. As with the other coding, the researchers then discussed any potential differences until agreement. With this information we can see which prerequisite skills are assessed by each question in the BDSI, and to what extent each question may differentiate between difficulties with prerequisites compared to course topics. The results from this coding is in Section 5.4.

Finally, in order to verify that our codebook was sensible, and what areas are covered by the other sources presented previously, we mapped the codes created in this stage to the ACM 2013 Curriculum Guide, the Core Concepts and the Misconception Catalogue. For the first two, this mapping was also done independently by two researchers in a similar fashion to the coding. The mapping to the Misconception Catalogue was done by a single researcher. These mappings are presented in Section 5.3.1, Tables 5.1 to 5.6.

5.2.3 Solution Development 2 - Differentiated Assessments

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

distilled a number of patterns and principles that can be applied to other assessments to make them differentiated.

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

This phase was conducted in a similar manner to the previous one. First, two researchers set out to modify one question (the question in M.2). This resulted in a number of patterns, only some of which were used, that were later generalized into the patterns and principles in the next phase. After this, other pairs of researchers modified other questions in the same way with the help of the previously identified patterns, thus validating and refining the patterns from the previous iterations. After a number of iterations of this process, modifying a number of questions, we arrived at the modified questions in Appendix M, some of which are presented in the main body of this chapter in Sections 5.5.2 to 5.5.4.

5.2.4 Reflection to Produce Principles - PAPRIDA

In this phase, we examined the patterns and our experiences using them to create differentiated assessment (above), and generalized them into several patterns and principles for making assessments more differentiated, called PAPRIDA (PAtterns and PRinciples for Differentiated Assessment). The start of this phase essentially took place alongside the previous phase in the form of collecting patterns and associated experiences while modifying questions. Members of the research team first modified a few questions then proposed a number of potential patterns. These patterns were then

explored by other researchers in the context of other questions, where they were refined, and new patterns were suggested. When all questions were modified, these patterns and the experiences related to them were collected and generalized to produce the list presented in Section 5.5.1 (as a preview: Show your work, Asking for details, Altering terminology, Renaming variables, Adding another instance, Adding prerequisite distractors, and Distractors with code). As these patterns and principles were refined throughout the assessment modification phase, the constant re-visiting and application of them serves as an initial validation of the patterns and principles. Note, however, that while PAPRIDA contains a set of useful patterns and principles for making questions differentiated, it is most likely incomplete as it is tied to the sample of questions we analyzed.

5.3 Results: Prerequisite Skills in Advanced CS Questions

To answer RQ1: "What prerequisite skills do advanced CS questions depend on?", we analyzed a number of existing assessments as described in Section 5.2. We present our codebook of basic PL knowledge and program tracing knowledge produced during our empirical work of analyzing the advanced questions, which is presented below. The main contribution is in our coding of each of the individual questions, which gives concrete examples of the issue we aim to address in this paper, and which prerequisite skills they depend on.

5.3.1 The Codebook

The codebook is presented in Tables 5.1 to 5.6 below. We divided the codebook into the following six groups for ease of navigation:

- Basic Notional Machine
- Loops
- Values and Types
- Functions
- Objects

- High Level Skills

Table 5.3: The Codebook: Values and Types

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

Values and Types (continued)			
Skill	**ACM Knowledge Unit**	**Core Concepts**	**Misconception Catalogue**
Arrays: Declaring and indexing arrays, and what happens when the index is out of bounds.	SDF/Fundamental Data Structures/Arrays	AR2 (Understanding the difference between a reference to an array and an element of an array), AR3 (Understanding the declaration of an array and manipulating an array)	Misc: "Confusion between an array and its cell."

Table 5.4: The Codebook: Functions

Functions			
Skill	**ACM Knowledge Unit**	**Core Concepts**	**Misconception Catalogue**
Parameters: Being able to declare and use function parameters to pass data into a function.	SDF/Fundamental Programming Concepts/Functions and parameter passing	PA1 (Understanding the difference between call by reference and call by value semantics), PA2 (Understanding the difference between formal parameters and actual parameters)	Sorva's "Methods" Topic is more related to OO design issues than function/method calls. There is a separate Calls Topics that deals with these issues.
Return values: Being able to declare and use function return values (or output parameters) to pass data out of a function. "Void" return values are included here.	SDF/Fundamental Programming Concepts/Functions and parameter passing		Calls: "A function (always) changes its input variable to become the output."

Functions (continued)			
Skill	**ACM Knowledge Unit**	**Core Concepts**	**Misconception Catalogue**
Return: Being able to use "return" to stop executing a function and return to the caller.	SDF/Fundamental Programming Concepts/Functions and parameter passing		
Function scoping and data flow: Being able to understand the scoping of local variables in a function, that the local variables are not shared between different invocations of the same function, and where parameters and return values fit in the model.	SDF/Fundamental Programming Concepts/Functions and parameter passing	PA3 (Understanding the scope of parameters, using parameters in procedure design), SCO (Scope)	
Recursion: The ability to understand and utilize recursive function calls, how execution flows through recursive functions and how values from operations are processed and stored. This is essentially only a special case that requires a better understanding of the three previous skills: *parameters*, *returns* and *function scoping and data flow*. It is however useful to separate this skill from the others, as recursion in itself is often problematic.	SDF/Fundamental Programming Concepts/The concept of recursion	REC (Recursion)	Rec: a Topic of its own.

Table 5.1: The Codebook: Basic Notional Machine

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

Table 5.2: The Codebook: Loops

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

Table 5.5: The Codebook: Objects

Objects			
Skill	**ACM Knowledge Unit**	**Core Concepts**	**Misconception Catalogue**
Classes/records/ADT: Being able to declare classes or records. This means to have understood their role in providing a *template* for instantiating objects which took part in the computation. It should be clear that classes define new (user-defined) types and they are units of encapsulation and scope, but not, for example, a way of ordering method calls (the order of method declaration is not relevant even w.r.t. their forward use in most OO languages).	SDF/Fundamental Data Structures/Records and structs (heterogeneous aggregates), SDF/Algorithms and Design/fundamental design concepts and principles, encapsulation and information hiding, PL/Object Oriented Programming, Definition of classes, methods and constructors, PL/Basic Type Systems/Compound types built from other types	CO (Classes and objects)	OtherOOP: "An object is just a record."
Object/instance/variable: Being able to differentiate between a class and an object (i.e., an instance of the class, or an instance of a type), or different instances of the same class. This involves being able to distinguish which objects (i.e., instances of a class or other memory entities of a built-in type) are active at a given point of a computation, in particular which objects have different identity (and, possibly, state) although they have the same type. The difference between references equality and object equivalence, the understanding of the problems of deep copies of complex objects is another important OOP basic skill. See also "Values and Types".		CO (Classes and objects)	ObjClass: "Confusion between a class and its instance."

Objects (continued)			
Skill	**ACM Knowledge Unit**	**Core Concepts**	**Misconception Catalogue**
Object scoping and data: Understand the lifetime of members of objects in relation to the object as a whole and the program. Class creation strategies (syntactic and semantic details can be quite different among different languages) define the lifetime of instance variables and methods (collectively known as *members*). A student should be able to differentiate among static visibility rules (`public`/ `protected`/ `private`/package in Java) and the accessibility of a specific entity, since even a private member can be reached by holding a reference returned by a method.		SCDE (Scope Design, understanding difference in scope between fields and local variables, appropriately using visibility properties of fields and methods, encapsulation)	ObjState: "During a method call, an object attribute is duplicated as variable. The local variable is initialized from the updated by the method, then returned to object at"
Static: Understand the difference between static and non-static members of a class. Member objects can be shared among the instances of the same class[3]. The role of `static` (to use the Java lingo) members and their special initialization rules should be well understood.		STAM (Static variables and methods)	

Table 5.6: The Codebook: High Level Skills

High Level Skills			
Skill	**ACM Knowledge Unit**	**Core Concepts**	**Misconception Catalogue**
Coding style: Understand and utilize elements of the language to make it easier to understand and reason about the code. Involves for example, comments, naming, use of empty lines, indentation, etc.	SDF/Development Methods/Documentation and program style		

[3]Sometimes even among user defined sets of classes, for example in some Smalltalk systems.

High Level Skills (continued)			
Skill	**ACM Knowledge Unit**	**Core Concepts**	**Misconception Catalogue**
API usage: Being able to find and use functions in some library (e.g., the standard library of the language). This involves searching for relevant functions, and reading the documentation to understand the semantics.	SF/Cross-Layer Communications/Programming abstractions, interfaces, use of libraries		
Problem decomposition: Being able to decompose a larger problem into smaller pieces with known solutions. For example, figuring out that a particular problem can be expressed in terms of a graph and use an appropriate graph algorithm to solve the problem.	SDF/Algorithms and Design/Fundamental design concepts and principles, Abstraction, Program decomposition	DPS1 (Design and problem solving 1, understands and uses functional decomposition and modularization), APR (Abstractions/Pattern Recognition and Use)	
Reasoning about constraints: Being able to reason about what is known and what is not known about the specification (i.e., pre-conditions) and reason about their implications on a particular piece of code or a particular method of solving a problem. For example, the array is not sorted, so using binary search is not possible without sorting the array first. Therefore, a linear search is faster.		DPS2 (Design and problem solving 2, Ability to identify characteristics of a problem and formulate a solution design)	

High Level Skills (continued)			
Skill	**ACM Knowledge Unit**	**Core Concepts**	**Misconception Catalogue**
Meta-tracing knowledge: Knowing you need to go through some algorithmic process step by step to check an answer, executing/keeping track of a representation of computation. Knowing when you need to use an external representation (and the representation is good enough). Knowing where your limits are. For example, if a problem says: "trace this code", it probably does not involve meta-tracing since the problem explicitly tells the student to trace the code.		SVS (Syntax vs. semantics, understanding the difference between a textual code segment and its overarching purpose and operation)	

These tables also contain the mapping between our codes and the ACM 2013 Curriculum Guidelines, the Core Concepts and the Misconception Catalogue including an example misconception of each category.

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

5.3.2 Coding of Assessments for Prerequisites

Tables 5.7 and 5.8 contain our coding of our initial 11 assessments (Table 5.7) and the 13 questions in the BDSI (Table 5.8). The full text of these questions is in Appendices M and O.

Table 5.7: The results of our qualitative coding of prerequisites assessed in the questions from advanced courses shown in Appendix M (questions that were eventually modified) and Appendix O (questions that were only used to devise the coding).

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

Table 5.8: The results of our qualitative coding of prerequisites assessed in the questions in the BDSI. Questions B.6 and B.8 were eventually modified and are therefore available in the Appendix M.5 and M.6.

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

Questions varied from having no prerequisites to having many. Overall, we found three broad types of questions:

The first type is assessments that **mostly focus on prerequisite skills**. Not surprisingly, a typical example of such assessments are pre-exams that are given to students in the beginning of a course in order to assess prerequisite skills. These typically cover skills that are almost entirely captured by our prerequisite skills coding. For example, see question O.2 in the Appendix.

The second type is the opposite: **almost exclusively assessing course topics**, meaning that almost none of the skills assessed by the question appear in our prerequisites codebook from Section 5.3.1. Many of these questions were found in the BDSI, and asked the student to reason about a data structure in higher-level terms; for example, see question M.6 in the Appendix. This type of questions has an empty column for prerequisites, as can be seen in column M.6 of Table 5.7. Other similar questions were found in a course on algorithm design, where the student was asked to reduce problems into suitable standard algorithms (Appendix O.4) (see our coding of its prerequisites in Table 5.7). It is worth pointing out that this does not mean that these questions do not have prerequisites; it only means that these prerequisites are not related to basic programming skills, but perhaps aimed more towards skills in algorithmic thinking, analytical thinking, logical reasoning, abstraction, or mathematics. They may also have prerequisites to actually learn the material, but the question itself does not require applying those prerequisite skills directly; for example, a question about the behavior of a data structure may not directly refer to code describing that behavior.

The final type of question assesses **both prerequisite skills and course topics, often requiring many from each**. This was perhaps the type of question most interesting to examine in the context of this research, as these typically have the property that if a student fails to answer a question, it is often difficult to attribute the failure either to the course topic or to a prerequisite skill. This does not, however, mean that such questions are undesirable. On the contrary, most questions of this type contained many prerequisites because they require the student to show that they are able to integrate the course topics into their prerequisite knowledge, and use it to solve some kind of real problem. Most of the questions we coded are of this type; for example see question M.5 and our coding of its prerequisites in Table 5.7.

5.4 Results: Is the BDSI Able to Diagnose Prerequisites?

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

Group 1 – Course Topics Only – B.4, B.9, B.13: These questions almost exclusively assess course topics (some of them assess meta-tracing knowledge as well). As such, they do not indicate problems with prerequisites (which was good in this case). For example, question B.4 asks

about the time complexity of certain operations of a linked list, which does not depend on any prerequisites.

Group 2 – No Differentiation – B.2, B.11: These questions contain both prerequisites and course topics, but none of the distractors allow drawing conclusions regarding whether a learner has difficulties with prerequisites or course topics.

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

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

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

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

5.4.1 No Differentiation: BDSI Question B.2

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

(a) Add a given element to the beginning of the linked list.

(b) Remove the last element from the linked list.

(c) Return `True` if the linked list contains a given element.

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

5.4.2 Mixed Differentiation: BDSI Question B.1

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

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

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

(a)
```
temp = new MyListNode(e)
tail = temp
```

(b)
```
temp = new MyListNode(e)
tail.next = temp
```

(c)
```
tail.next = e
tail = e
```

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

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

(a) A student picking this answer over the correct answer could have difficulties with *object/instance/variable* (e.g., a student who does not understand that there are difference instances of `MyListNode` will likely not see the point in linking them), *meta-level tracing* (in this case, all operations only using the tail reference would work, while others do not), or with linked lists (e.g., forgetting that all nodes need to be reachable from the head).

(b) A student picking this answer over the correct answer likely has difficulties with either *meta-tracing knowledge* (in this case, the implementation works for one insertion, but not for two), or with the course topic linked lists (e.g., not understanding the purpose of the tail reference).

(c) A student picking this answer over the correct answer likely has difficulties with either *types* or *object/instance/variable*. The solution is correct, except that both `tail` and `tail.next` are supposed to refer to a node rather than an element. This could either be due to the student not realizing this through reasoning about the types, or by not realizing that a new node instance needs to be created.

(d) This solution would be correct in a linked list without a tail reference. As such, a student picking this answer over the correct answer is likely able to trace and understand the code, while only having difficulties with linked lists (in particular, tail references).

5.4.3 Mixed Differentiation: BDSI Question B.3

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

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

(b) Author a singly linked list class of your own and test the timing of it against the timing of the unknown list for all methods. If the timing between your singly linked list and the unknown list is exactly the same for all methods, the unknown list is a singly linked list, otherwise it is a doubly linked list.

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

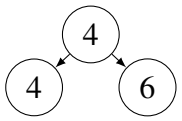
(d) None of the above experiments would be able to answer this question. You would need to be able to examine the code to determine if the implementation uses a `previous` reference.

Similar to question 2, most options involve both course topics (asymptotic notation and linked lists in this case) and prerequisites. As such, most distractors do not distinguish between the two.

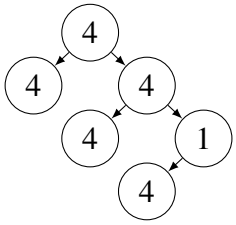
The distractor (a) is, however, interesting. It suggests that one can distinguish between a singly linked list and a doubly linked list by creating two instances of the unknown class, and doing the same sequence of operations to each of them in turn while measuring the time. For this to be true, the two instances of the class need to behave differently in some regard, which in turn implies some shared state between them. In particular, this would be true for a student who do not realize that different instances have a different set of member variables, and thus append all elements from the two instances to a single list. This reasoning implies that a student who picks the distractor (a) is likely to have difficulties with *object/instance/variable*. At the same time, the student might have an issue with the course topic by not understanding what doubly-linked list means (as if the lists might be linked to each other "doubly").

5.4.4 Differentiation via Triangulation: BDSI Question B.7

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



`sum_leaves` should return 10



`sum_leaves` should return 12

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

(a)
```
DEFINE sum_leaves(node)
  value = 0
  IF node.left == nil AND node.right == nil THEN
    value = value + node.item
  ENDIF
  IF node.left != nil THEN
    sum_leaves(node.left)
  ENDIF
  IF node.right != nil THEN
    sum_leaves(node.right)
  ENDIF
  RETURN value
ENDDEF
```

(b)
```
DEFINE sum_leaves(node)
  IF node.left == nil AND code.right == nil THEN
    RETURN node.item
  ENDIF
  IF node.left != nil THEN
    RETURN node.item + sum_leaves(node.left)
  ENDIF
  IF node.right != nil THEN
    RETURN node.item + sum_leaves(node.right)
  ENDIF
ENDDEF
```

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

(d)
```
DEFINE sum_leaves(node)
  IF node == nil THEN
    RETURN 0
  ELSE IF node.left == nil AND node.right == nil THEN
    RETURN node.item
  ELSE
    RETURN sum_leaves(node.left)
         + sum_leaves(node.right)
  ENDIF
ENDDEF
```

In this "select all correct" multiple choice question, we explored all combinations of distractors selected by the student, with the idea the combination might be used to better pinpoint which prerequisites (or course topics) that might be problematic:

(a) *return value* or *recursion*

(b) *return*

(c) *meta-tracing knowledge*

(d) -

(a,b) *return* and *function scoping*

(a,c) *recursion* and *meta-tracing knowledge*

(a,d) *function scoping*

(b,c) *return* and *loops*

(d,b) *return*

(d,c) *meta-tracing knowledge*

(a,b,c) Likely *conditionals*

(a,b,d) Likely *recursion*

(b,c,d) You know *function scoping* and *return values*, but not *loops* and *return*

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

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

5.5 Results: Differentiated Assessment Examples and Principles

To answer RQ3: "What are examples of differentiated assessments and principles for designing differentiated assessments?", we utilized our prerequisite coding from (Section 5.3) and modified a number of the analyzed questions to make them able to differentiate between difficulties with prerequisite skills and course topics. During these modifications, we also collected a set of patterns and principles to do these modifications, which we later refined into PAPRIDA (PAtterns and PRinciples for Differentiated Assessment). This section presents the patterns and principles, followed by a detailed presentation of the analysis and modifications to one question from each of the three course topics: advanced object-oriented programming, data structures and concurrency.

Each of these questions is examined in detail. For each question, we present what course topics the question is designed to assess, along with some additional background. After that, we describe the steps taken to modify the question: first the question was analyzed to find prerequisites. Then, a number of those were selected to be assessed explicitly. There was a trade-off between assessing all prerequisites explicitly in a way that it is possible to diagnose between all of them versus increased size of the assessment. Finally, we applied a number of patterns and principles from PAPRIDA to make the modifications to the question. Thus, these examples can be seen as "worked examples" of how to apply PAPRIDA to improve a question, either by making the question explicitly assess the prerequisites, or by introducing new prerequisites that are explicitly assessed to a question. Additional questions we modified are reported in Appendix M.

5.5.1 PAPRIDA: PAtterns and PRinciples for Differentiated Assessment

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

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

Asking for details: One alternative to the previous approach is to add a small question that asks the student for some specific detail of the code in the question related to some prerequisite skills. This could, for example, be to ask the student to point to lines in the code that accesses a particular part in memory or asking for the type of a particular expression in a particular context. We used this kind of technique in the concurrency exercises to assess whether students know when *Indirection* occurs. The benefit of this approach is that it does not increase the grading time by much, while still giving an indication of the prerequisite skills,

but it might be difficult to find a small but precise enough such question. Another option is to insert a print statement that outputs something very specific and ask about that.

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

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

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

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

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

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

5.5.2 Example Assessment for Advanced OOP on Inheritance and Polymorphism

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

The question provides an implementation of the classes depicted in Figure 5.2 in Eiffel[4], and the code below in Listing 5.1 that creates instances of the classes and calls the member function `eat` in various ways. The student is then asked whether different calls to `eat` is a compile-time or run-time error, and are then asked to add some constraints (the full question is in Appendix M.4).

Listing 5.1: The main part of the Eiffel code for assessment on Polymorphism (M.4)

```
1  class APPLICATION
2  create
3    make
4
5  feature -- Main
6
7    make
8        -- Run application.
9      local
10       a: ANIMAL
11       c: COW
12       g: GRASS
13       f: FOOD
14     do
15       create c
16       create g
17       a := c
18       f := g -- focus on this
19       print (a.out + " is going to eat: " + f.out + "%N")
20       a.eat (f)
21     end
```

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

```
22
23 end
```

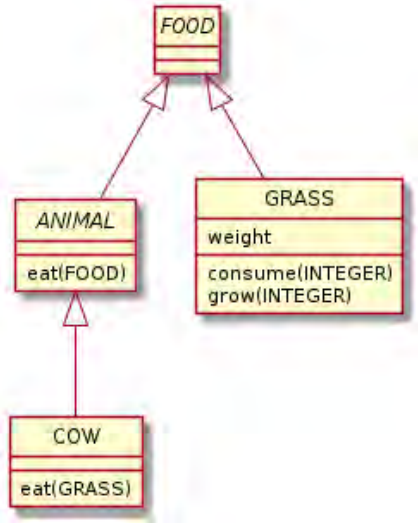


Figure 5.2: UML Class diagram for the polymorphism assessment

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

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

To guide our modification of the question, we considered which prerequisites were most essential and conceptually related to the advanced skill of polymorphism. In particular, a clear understanding of references, the impact of their types, the handling of method parameters, and the ability to trace the flow of the computation are key for solving the exercise. Thus, it could be useful to add a couple of specific questions before the advanced part of the question, to reveal weaknesses or misconceptions in prerequisites. We decided to add tracing questions as that has congruence with the original exercise's first step on run-time and compile-time error identification; these added questions targeted checking the basic OOP notional machine was solid enough to support the new OOP concepts. To avoid the question growing overly long, we did not add more.

To enable some of our modifications, first we modify the `out` methods to print the dynamic type of an object (see Appendix M.4 for the full source code of the classes); this can be leveraged in tracing questions. We then made the following modifications (see the main modified question code below in Listing 5.2):

1. Add a new concrete `FOOD` class (e.g., `PLANKTON`) and create an object `p` from this class (*Adding another instance*, a class in this case, from Section 5.5.1)

2. Add a method `log_food` with a parameter `x` of type `FOOD`, the method just prints the dynamic type of the actual parameter bound to `x` (*Asking for details* from Section 5.5.1)

3. Add a question about the output of `log_food(p)`, `log_food(g)`, `log_food(f)`, where `g` is a reference to a `GRASS` object and `f` is a reference to a `GRASS` object of `FOOD` static type. Note that it would not be legal to call `log_food` with `f` as an actual parameter before assigning it to a concrete object. In order to do this, the type of the parameter must be marked as `detachable`[5]. This observation can be used to assess the students' understanding of nullable references by asking why the call to `log_food2` at line 18 is legal while `log_food` is not. (*Asking for details* from Section 5.5.1)

Listing 5.2: The modified Eiffel code for the modified assessment on Polymorphism (M.4). Additions are highlighted.

```
1  feature -- Main
2
3    make
4        -- Run application.
5      local
6        a: ANIMAL
7        c: COW
8        g: GRASS
9        f: FOOD
10       p: PLANKTON
```

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

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

5.5.3 Example Assessment for Data Structures

In this section, we describe our modifications to a data structure exam problem. The problem is available in Appendix M.1, but we provide a summary here. The question presents students with the code in Listing 5.3 and are asked to:

(a) determine whether it implements a stack, a queue, a priority queue or a union find data structure

(b) implement a suitable `size` method (from four options)

(c) determine which out of four possible invariants are upheld by the data structure

(d) trace the behavior of a sequence of insertions and removals

(e) reason about the number of array accesses the `remove` method performs in the worst case

(f) reason about the number of array accesses the most expensive public operation perform in the worst case

(g) reason about a generic sequence of operations

(h) reason about memory consumption of the data structure

Listing 5.3: Code from the data structure question (M.1)

```
1  public class Y<Key extends Comparable<Key>>
2  {
3      private Key[] A = (Key[]) new Comparable[1];
4      private int lo, hi, N;
5      public void insert(Key in)
6      {
7          A[hi] = in;
8          hi = hi + 1;
9          if (hi == A.length) hi = 0;
10         N = N + 1;
11         if (N == A.length) rebuild();
12     }
13     public Key remove() // assumes Y is not empty
14     {
15         Key out = A[lo];
16         A[lo] = null;
17         lo = lo + 1;
18         if (lo == A.length) lo = 0;
19         N = N - 1;
20         return out;
21     }
22     private void rebuild()
23     {
24         Key[] tmp =
25             (Key[]) new Comparable[2*A.length];
26         for (int i = 0; i < N; i++ )
27             tmp[i] = A[(i + lo) % A.length];
28         A = tmp;
29         lo = 0;
30         hi = N;
31     }
32 }
```

The code included in the question implements a queue with a circular array and two integers: `lo` is the index of the first element in the queue and `hi` is the index just after the last element in

the queue. The variable `N` represents the number of element in the queue. The array is rebuilt with doubled size whenever the insertion of an element exhausts the capacity of the array.

The question assesses the following topics on data structures:

- to distinguish among different data structures;
- to understand code implementing a data structures;
- to know worst case and amortized complexity notion;
- to analyse the complexity of an algorithm expressed by a piece of code

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

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

By further examining the different items, we can see that items (a) to (d) require close inspection of the code to figure out how the data structure works, which in turn require skills related to code comprehension. For example, lines 8 and 17 can be used to rule out the possibility that the data structure is a stack for item (a), lines 10 and 19 can be used to identify that `N` is indeed the number

of elements in the stack for item (b), and understanding the circularity and the rebuilding policy are essential for answering (c) and (d). Items (e) and onwards are then more focused on course topics (in this case, mostly asymptotic analysis) while relying on prerequisites to a lesser extent.

By looking at those critical lines, we decided the most critical prerequisites were: operators (modulus in this case), conditionals, arrays and array iteration. Even if the learner understood conceptually the data structures in part (a) (stack, queue, priority queue, or union find), without those prerequisites they'd have difficulty making sense of the implementation code. As those prerequisites are not assessed explicitly anywhere in the question, and are deemed critical, we focused our modifications on making these prerequisites explicit. We considered making some changes to the data structure class itself to bring out some prerequisites by printing values, but did not as the cost of increasing the code complexity and size would distract from the core task for the learner. Our changes to accomplish this are outlined below. The full modified question is presented in Appendix M.1.4.

- A new item was added at the end of the tracing task in item (d), that asks how many times the rebuild method is called. This assesses the prerequisite knowledge on conditionals in order to check that the building policy has been understood (*Asking for details* from Section 5.5.1). This addition balanced low added learner time cost versus helping instructors to cluster incorrect answers due to conditional prerequisite issues; as assessment designers we found it important to consider that trade-off. In this case, it seemed reasonable, but another good effect was the addition also helps scaffold the learner's focus onto a critical part of the code for the advanced concepts (worst-case analysis).
- Another concrete tracing task was added after part (c) to assess knowledge on operators (modulus), arrays (indexing and storage), and array iteration (*Asking for details* from Section 5.5.1).
- An item was added that proposes a specific input situation and asks to establish if it can occur after a sequence of insert and remove call. This requires the students to reason about the pre- and post-conditions and the invariants of the code (*Asking for details* from Section 5.5.1).
- We finally add an item to assess knowledge about values and references. Additionally, this exercise also assesses the ability to differentiate between an object/ADT and its instances, which was deemed an important and related prerequisite skill, even though it was not strictly necessary to answer the original question. More precisely, students are asked to establish the values of variable `a` and `b` after executing the following piece of code. (*Adding another instance/Introducing aliasing* from Section 5.5.1):

```
1 Y y = new Y();
2 Y z = new Y();
3 Y w = z;
4 w.insert(3);
5 z.insert(1);
6 y.insert(2);
```

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

5.5.4 *Example Assessment for Concurrency/Synchronization*

The analyzed questions assessing concurrency were designed to explicitly assess *threads*, *busy-wait*, and the student's ability to use suitable synchronization primitives (out of *semaphores*, *locks* and *condition variables*) to solve synchronization issues. This can be quite clearly seen from the original question in Appendix M.2. The question presents students with the code in Listing 5.4, which implements a buffer containing strings, and asks students to identify and solve occurences of *busy-wait*, and then to identify and solve any remaining synchronization issues.

Listing 5.4: Code for the concurrency exercise (M.2)

```
1  struct idea_buffer {
2    // All ideas in the buffer. Empty elements are
3    // set to NULL.
4    const char *ideas[BUFFER_SIZE];
5    // Number of ideas in the buffer.
6    int count;
7  };
8  // Add a new idea to an empty location in the
9  // buffer. Returns 'false' if the buffer is full.
10 bool idea_add(struct idea_buffer *buffer,
11                const char *idea) {
12   int found = BUFFER_SIZE;
13   for (int i = 0; i < BUFFER_SIZE; i++) {
14     if (buffer->ideas[i] == NULL) {
15       found = i;
16       break;
17     }
18   }
19   if (found >= BUFFER_SIZE)
20     return false;
21   buffer->ideas[found] = idea;
22   buffer->count++;
23   return true;
24 }
25 // Get and remove a random element from the
26 // buffer. If no elements are present, the
27 // function waits for an element to be added.
28 const char *idea_get(struct idea_buffer *buffer) {
29   while (buffer->count == 0)
30     ;
```

```
31   buffer->count--;
32   int pos = rand() % BUFFER_SIZE;
33   while (buffer->ideas[pos] == NULL) {
34     pos = (pos + 1) % BUFFER_SIZE;
35   }
36   const char *result = buffer->ideas[pos];
37   buffer->ideas[pos] = NULL;
38   return result;
39 }
```

By analyzing the question, we can see that it also requires the student to understand the following prerequisite concepts, even though they are not explicitly assessed:

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

By examining the study by Strömbäck et al. [Strömbäck et al., 2019], which studied students' performance on this particular question, we can see that the authors observed that it was not always clear if particular categories of incorrect answers were due to students not understanding

concurrency, or had some misconceptions regarding different instances and/or pointers. The authors also noted that some students attempted to synchronize local variables in functions, suggesting that students either do not understand what is relevant to synchronize, or that students do not understand function scoping. Because of this, we want to explicitly assess the skills *Objects*, *Values and references*, *Indirection* and *Function scoping and data flow* in order to differentiate difficulties in prerequisites from difficulties with the course topics. In general, as assessment designers, it may be useful to do or draw on qualitative observation of students responding to the question, to aid in identifying and focusing on difficulties seen in practice.

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

- The name of the pointer variables referring to the shared `idea_buffer` were altered in order to make it impossible to rely on pattern-matching to realize that the variables inside the struct may refer to the same variable, thus requiring the student to understand how pointers work in order to be able to identify shared data. This is explicitly assessed by adding the following question (*Renaming variables* from Section 5.5.1):

  After executing the following code, what is the value of the variable `res`?

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

  While that shared name cue can help some students with reference weaknesses complete this problem early on in the course, as assessment designers we considered this early topic (busy wait) a good time to explicitly assess and raise issues with that prerequisite, which will cause problems with more complex data structures later in the course.

- In order to assess the *object* skill explicitly, we add the following question that requires the student to be able to differentiate between different instances of the same struct (*Adding another instance* from Section 5.5.1):

  What is the expected behavior when executing the last line in the code below?

```
1 struct idea_buffer a, b;
2 idea_init(&a);
3 idea_init(&b);
4 idea_add(&a, "a");
5 idea_get(&b); // <-- here?
```

- In order to assess the *indirection* skill explicitly, we added a question asking the student to mark all locations where data inside an `idea_buffer` is accessed (*Asking for details* from Section 5.5.1).

- Finally, in order to assess the *function scoping and data flow* category, students are asked which variables are not shared between threads. If students fail to mark any of the variables, it indicates that the student might incorrectly believe that local variables are shared between threads, and thus that the student might not understand function scoping properly (*Asking for details* from Section 5.5.1).

By adding these parts to the question, the student may get some help examining the code for things relevant to the synchronization tasks, but more importantly, the instructor may be able to examine the answers to these parts and assess whether any mistakes in other parts of the question are due to concurrency or not. The assessment could even have the students email their TA or submit online their answers to the added first two parts, to get quick feedback on prerequisite weaknesses and remedial help, before struggling on the advanced task.

As assessment designers we also could have removed some prerequisites in the question, at the expense of making the code more verbose. For example, the modulus operator is used in an idiomatic way on line 46 and 48, to wrap the array index around the end of the array. The question already mitigates this by providing a code comment describing what the code does, so we did not make that change.

5.6 Limitations

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

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

Qualitative analysis by experts. The qualitative coding of prerequisites required for questions relied on our understanding of students' mental models and informally remembered behavior of students on questions. It was not an empirical study of students' behavior. Our understanding may

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

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

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

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

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

5.7 Discussion

First, we summarize our results, then we discuss our results for each research question more deeply. We then discuss how making *differentiated assessments* raises questions for theories of computing knowledge and how we might refine such theories by analyzing assessments. Lastly, we discuss

other areas of future work, including applying our paper's approach to other course topics, and we recommend empirical validity studies for our initial differentiated assessments.

5.7.1 Summary of Results by Research Question

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

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

For RQ2 "To what extent can an existing concept inventory for data structures with a validity argument – the BDSI [Porter et al., 2019] – also diagnose difficulties with prerequisite skills?", we found even for high quality questions, many unable to precisely diagnose prerequisite skill issues from incorrect answers. Of the BDSI questions, ten required prerequisites, and one of those ten was diagnostic (see Section 5.4.4), five had some distractors indicating a small set of prerequisites and/or advanced skills, and four could not diagnose (Groups 2 and 4 in Section 5.4).

Thus, for RQ3, our goal was to investigate the feasibility of designing assessments that are able to differentiate between the two, ideally with no or minimal increase in the time required neither for students to do the assessments, nor for teachers grading the assessments. We then made six modified assessment questions, designed to better differentiate (according to our judgement as instructors), and distilled initial patterns and principles for differentiated assessments (PAPRIDA), including computing-specific patterns like "introducing aliasing" in Section 5.5.1.

5.7.2 Research Question 1

While prior work has found prerequisite weaknesses seem to impede learning in advanced courses [Valstar et al., 2019, Fisler et al., 2017] and categorized types of advanced skill assessments [Simon et al., 2010], we contributed the first analysis of advanced skill assessments' prerequisite dependencies in the computing education literature. Most questions we analyzed required some prerequisite knowledge. For example, question M.1's course topic is data structures, but learners might get parts of it incorrect due to not being able to trace the loop inside the `rebuild` function. In this case, it might be due to a lacking understanding of loops, or a lacking understanding of operators (modulo in this case), which could make the student believe that the function does something more advanced than resizing the data structure's array.

Some questions did not require any of the prerequisites we analyzed. These interesting questions were more conceptual, for example, some high level data structures questions on the BDSI (see, for example, question B.4 in Table 5.8). These questions can be useful for finding and correcting gaps in conceptual knowledge. Future work should find ways to make such focused and specific questions, contributing general patterns and ones specific to computing education specific.

While it can be good to have such questions, assessments that only cover the advanced topics are not necessarily better. First, students might be able to answer such questions by rote memorization,

seemingly knowing part of the course topic without knowing prerequisite skills; for example, for a question asking for the big-O runtime for an algorithm, a learner may answer correctly without understanding how to derive it from the algorithm or what it really means. Second, if we mostly use questions solely on the course topics, we might teach and assess course topics in an isolated way, without practicing the prerequisites you need to use them. Without practice learners can forget or become weaker in those prerequisites over time.

5.7.3 Research Question 2

Prior work in computing education has made advanced assessments without analyzing their feasibility for diagnosing learners. The results of our qualitative analysis of the BDSI distractors showed few questions could precisely diagnose prerequisites. Given our results, concept inventories for course topics should not be presumed useful for diagnosing prerequisites. This is not surprising, since this is not a design goal of the BDSI or concept inventories in general; nor do they usually desire to make validity arguments that individual questions have validity. However, given the difficulty in differentiating between advanced and prerequisite skills, existing advanced assessments may have difficulty diagnosing advanced skills also, since the cause can come from either advanced or prerequisite weaknesses. This connects to related work in educational assessment we described in Section 2.2.3; one should not assume concept inventories and other assessments have desirable properties, such as diagnosing learner's misconceptions [Jorion et al., 2015, Sands et al., 2018]. Computing education assessment research may benefit from considering formative use as part of validity studies and the initial design work of questions for advanced assessments.

5.7.4 Research Question 3

Given that prior work has focused on making early or advanced skill assessments, the main practical contribution of this work is the idea of differentiated assessments, examples, and a toolbox of patterns (listed in Section 5.5.1) for augmenting assessments to be more differentiated. We have found a few initial computing specific technical tricks, like "Introducing aliasing" or "Renaming variables", that have the underlying goal of testing which chunks of prerequisite knowledge the solver is able to use in a new, more advanced, setting. These strategies are particularly useful when a large number of students are expected to take the modified assessment, as it may require less manual grading compared to, for example, "Show your work" questions, while still being able to diagnose some prerequisites issues. For example, as in the concurrency question in Section 5.5.4, one can carefully re-structure code in an assessment and/or add question parts that highlight the relevant prerequisites. This restructuring can enable good feedback on prerequisites without increasing the workload for the student considerably, and also may be easy to grade automatically and thus give immediate feedback to students. At the same time, our PAPRIDA recognize patterns teachers have used for millennia to get students to reveal their problem solving process, like "Show your work" and "Asking for details".

The idea of augmenting assessments to be more differentiated is different from a pre-exam at the start of an advanced class: in fact, students have probably *passed* such an exam, but this is in many

cases not enough to guarantee that prerequisite skills are at the level required to focus properly on the new ones. Differentiated assessments can be more narrowly focused on what is actually instrumental to course topics, and this might also help students in making sense of what they learn.

5.7.5 Implications for Instructors and Teaching

Our design and envisioning work suggests there is much potential for differentiated assessments to help instructors and students. For example, an early set of differentiated questions might be "disguised" as quizzes on the first few lectures, which gives most of the benefits of a pre-exam, while also giving the students the opportunity to practice on course topics. In addition to keeping students motivated to take the test, this could utilize the instructors' valuable time with students more efficiently. Assessing prerequisites explicitly in midterms and final exams may help catch cases where a student is not proficient enough with the prerequisites to be able to focus on the course topics. Instructors might also find common weaknesses in the specific prerequisite knowledge, may realize why their tests are too hard and try to make teaching that more explicit and give more feedback to students.

Differentiated questions seem especially promising for improving equity. The more students come in with varied backgrounds, the more potential benefit for both identifying students falling behind and helping them with targeted feedback, especially in more advanced degree programs. While promising, this benefit should be empirically studied with actual students.

In making differentiated assessments, our design work showed it can be infeasible to assess all the prerequisites diagnostically; this might be mitigated by giving follow-up pre-exams. Instructors making differentiated assessments should prioritize, by asking TAs to gather common difficulties or drawing on their experience in class and office hours when choosing which prerequisites to prioritize. While our design work suggests individual questions cannot cover everything, differentiated assessments can work with traditional pre-exams. If a differentiated assessment is able to tell that *some* prerequisite is lacking, the student can be instructed to take a more extensive test. This could check all prerequisites in depth, which can direct the student to further practice.

Instructors might also benefit by following our work's process of analyzing and modifying their existing questions to make them more differentiated. Instructors might surface and reflect upon the effectiveness of their assessments and their inclusion of prerequisite skills, intended and unintended. Using this information, the teacher might improve their teaching, via a more informed decision of what prerequisites might be beneficial to assess explicitly and address, or perhaps which could be excluded from the assessment. Instructors teaching an advanced class could go through that process together within or across institutions. Given our lack of in-depth empirical studies with students, we advise instructors to try their modified questions with a small set of students before rolling them out to their class (as with any assessment change).

5.7.6 Future Work

In this section, we outline empirical validity studies for our theoretical findings presented in this report. We also discuss how our work leads to other interesting research topics for future

investigation, which our community can pursue, such as making differentiated assessments for other advanced topics and prerequisites, and new patterns and principles for designing differentiated assessments.

5.7.6.1 Evaluating questions empirically for validity properties, especially for formative use

As mentioned in our limitations (Section 5.6), the modified questions have not been empirically evaluated with students. A validity argument based on, for example, Kane's framework [Kane, 2013, Nelson et al., 2019] should be made using empirical studies, such as think-alouds and using the questions in actual learning environments.

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

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

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

5.7.6.2 Creating differentiated assessment questions for each advanced computing topic and for different prerequisites

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

This future work should evaluate the generality of our PAPRIDA, the patterns and principles presented in Section 5.5.1. Our paper's design method can be applied to different questions for advanced courses, both inside and outside of the topics covered in this report. This would involve coding a question according to the codebook in Section 5.3.1, determine which prerequisites are relevant to assess, and applying the patterns in Section 5.5.1 to make them explicit, and ideally empirically evaluate the resulting questions. This work may also evaluate how well our PAPRIDA generalize, and contribute new PAPRIDA. There are likely computing education specific question patterns for each area of knowledge, just waiting to be discovered and distilled.

5.7.6.3 Analyzing curricular assumptions using prerequisite coding of assessment questions

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

5.7.6.4 Extending our paper's assessment design method to also include qualitative coding of advanced course topics

In this paper we only code skills that were considered prerequisites to at least one of the advanced topics. It would be useful to extend our method to include some advanced skills as well, using the method presented in Section 5.2.2. This would extend the codebook to include codes for advanced skills. Such an extension would also allow exploring curricular assumptions from earlier advanced courses to later advanced courses (see Section 5.7.6.3).

5.7.6.5 Developmental stages for course topics

Another interesting extension to the framework is to introduce *developmental stages* for the course topics. These could be based on the coding of course topics as mentioned above, but also from other observations in the literature. For example, in a course on concurrency, we can conjecture that a student start by being able so solve simple, explicit synchronization goals, then progresses to be able to identify some shared data and synchronize it properly at a coarse level, and finally at a finer level. Given these developmental stages, one could then create a matrix with developmental stages on the horizontal axis, and a set of tracing weaknesses from Section 5.3.1 that are known to be problematic in the context (if not all of them). Then, for each cell in the matrix, one would write what a hypothetical student with the given tracing weakness at the given developmental stage would be able to do, and what the student would not be able to do. This information could then be used to select appropriate questions during a course, or to design questions that are able to diagnose at what developmental stage a particular student is at, and which, if any, tracing weaknesses the student have.

Chapter 6

CONCLUSION

This chapter contains a summary of the thesis statement, a list of contributions made with respect to the thesis statement, then discussion and future work.

6.1 Summary

In this thesis I have demonstrated *a new theory of programming language knowledge that includes mappings from syntax to semantics and their nested combinations can serve as the basis for more granular tools for learning and more precise assessments of that knowledge.*

I started by creating a **theory of basic programming language knowledge that includes the mapping from token-level syntax to semantics and machine state**. For example, in `var x=1;` the `x=` assigns a value to `x` in the variable assignment table in the machine state. The key motivation was to decompose that PL knowledge into small granular parts, to enable learning via human causal inference. See Section 3.1 for more detail.

To create a new **more granular tool for learning** teaching that knowledge, I used that theory as the basis for creating new instructional and curriculum design for that knowledge, and implemented that in PLTutor. I created an instructional design for showing those mappings, by stepping through granular program execution steps while consistently showing each part of that mapping: token-level syntax highlighting, machine state highlighting, and semantics as a functional form (such as a natural language explanation) followed by the change from one execution step to the next. Then I created a reading-first spiral curriculum design and enabled granular teaching of that knowledge by adding a teaching layer of abstraction on top of program visualization systems, which supports mixing conceptual instruction, program visualization, and scaffolded assessments. For more details, see Section 3.2.1 for instructional design and Section 3.2.2 for curriculum design and flexible teaching granularity.

To design **more precise assessments**, I then extended my theory of basic PL knowledge in two different ways. In Chapter 4 I included nested combinations of that knowledge. For example, one combination is a nested `if` statement where the outer `if` condition is true and the inner `if` condition is true; another separate combination covers the outer condition is false and inner condition is true. I then created a systematic test of those nested combinations, by generating many questions that each tested a separate part of that knowledge.

In Chapter 5 I included relationships with more advanced programming skills, then created a new design genre of assessment question: *differentiated assessments*. Differentiated assessments expand the theoretical scope of traditional advanced assessment questions to also attempt to more precisely diagnose prerequisites (such as program tracing skills). For example, for an advanced

question on concurrency, new sub-questions were added to explicitly assess prerequisite scoping skills (see Section 5.5.4 for more detail). Chapter 5 also includes other examples and initial computing-specific design principles for differentiated assessments: show your work, asking for details, altering terminology, renaming variables, adding another instance, adding prerequisite distractors, and distractors with code (see Section 5.5 for more detail).

6.2 Contributions

This thesis makes these four contributions:

- a theory of programming language knowledge that includes the mapping between token-level syntax, semantics, and execution state, and nested combinations of that knowledge
- PLTutor, a program tracing learning tool with flexible teaching granularity that mixes conceptual instruction, program visualization, and scaffolded assessment questions, via a novel teaching layer of abstraction on top of program visualization systems
- a program tracing assessment that precisely and systematically assesses nested combinations of program tracing knowledge, for use as a formative assessment
- *differentiated assessments*: a new genre of advanced programming question designs made to precisely diagnose problems with advanced knowledge (such as concurrency) versus prerequisite skills, such as program tracing skills (like knowing scoping rules that determine what variables may need concurrency controls added), and initial examples and computing-specific design principles for differentiated assessments

6.3 Discussion

6.3.1 Curriculum

Learning programming is hard, and teachers and researchers have addressed these difficulties in diverse ways over the past 70 years; while these attempts appear diverse, they have been designed and evaluated in a writing-focused pedagogical context. My thesis work questions that foundational assumption of curriculum design in computing education. This thesis explored a different direction, from old pedagogical ideas about starting with higher-level program comprehension skills earlier in learning [Deimel and Moffat, 1982, Kimura, 1979]. My reading-first spiral curriculum teaches from small example programs to more meaningful larger programs; this curriculum innovation does for reading what the spiral approach in the 1970s did for writing: bridging the bottom-up approach and the whole program approach [Lemos, 1975, Lemos, 1979, Shneiderman, 1977]. For reading the bottom-up approach is the "ten fingers" curriculum of typing in lines of code into an interpreter and learning from the results [Bork, 1971] (from Bork in 1971[1]) and the whole program approach

[1]Implemented in an excellent modern way by Reduct [Arawjo et al., 2017] with more granular practice, and concreteness fading (which ultimately ties back to Mayer's work with natural language statements fading to syntax

starting from large useful programs from Kimura [Kimura, 1979], and Deimel and Moffat's work in 1980 [Deimel and Moffat, 1982].

Why was that curriculum gap unfilled for 40 years? One fundamental barrier was a technical gap to finding that middle ground: a lack of flexible teaching granularity for mixing instruction and practice from small parts of programs to larger ones.[2] Implementing a reading-first spiral curriculum required a teaching layer of abstraction that could target a teaching action to a small part of a program's execution within a larger one, in order to focus instruction and practice at the right grain size for novices, growing from sub-expression to larger parts of a program.

This layer of abstraction could allow learners to engage with large meaningful programs in a granular and authentic way; for example, in a class this might take the form of "Here's this large program that does this important thing in society. We're going to learn about this small part of it, towards learning how the whole thing works." The class or pairs or individuals might use a tool that varied that granularity to teach a part of the program. Once the program is understood, the curriculum could move to modification practice: "How might we change it for the better, and what consequences might that have?"

What kinds of new teaching layers of abstraction could enable integrating different kinds of granular instruction, practice, and feedback seamlessly within the same learning tool or IDE? They might support a flowering of new curriculum designs for computing education that mix approaches, further enabled by backend integrations of learning tools and learner models [Brusilovsky, 2004, Sirkiä and Haaranen, 2017, Brusilovsky et al., 2015, Stegeman, 2019].

Those layers of abstraction could enable curriculum design adaptation [Bimba et al., 2017] to fit the individual differences of the learner, or simply allow the learner to choose what they want. For example, the reading-first spiral approach may better fit learners that are more risk-averse and wish to start by methodically understanding how programs work. People with comprehensive information processing styles (which statistically associates with female gender identity) want to gather more complete information, and may also prefer to understand how a tool works instead of tinkering with the tool to figure it out (which novice writing practice can easily become) [Meyers-Levy, 1986, Burnett et al., 2016].

The long term effects of new curricula must also be evaluated, for benefits and potential unintended consequences. For example, in PLTutor's evaluation in Section 4.3 PLTutor users had more normally-distributed mid-term exam scores, but the maximum was lower (compare far right of 3.4.e & 3.4.f). While rare, the long tail of learning outcomes is extremely important at a societal level and must be analyzed empirically [Fortunato, 2017]; for example, a curriculum that halved the number of future Newtons and Einsteins could be quite bad.

[Bayman and Mayer, 1988, Mayer, 1982], and the earliest example to my knowledge in Knowlton's video on the L6 programming language in 1966 (click here to view at 10:44) [Knowlton, 1966]).

[2] And the lack of sub-expression stepping until ZStep in 1984 [Lieberman, 1984], though manually created animations at that level go back to at least 1975 [Baecker, 1975].

6.3.2 Theory and Design

Basic theories of computing knowledge remain ripe for discovery and refinement, even after 70 years of teaching and research. This thesis starts with extending theory of basic skills for programming, in particular for basic PL knowledge and program tracing, then builds upon that advance. In that vein, this thesis concretely raises the possibility that other skills in computing might be taught better by asking these research questions, framed with key ideas from this thesis:

- What are prior theories of this skill, and how might we extend them or create new ones?
- How might we teach at a useful level of granularity?
- How might we systematically assess the skill directly?
- How might we assess the skill as a dependency when assessing more advanced skills?

This thesis work also evolved alongside discourse about use of theory in computing education research; my work is an example showing the potential of using theory to invent new designs for curricula, learning environments, and new kinds of systematic assessments. This discourse says new ways of conceiving of what we need to teach (i.e. new theory) can lead to radically new ways to teach and assess [Nelson and Ko, 2018, Dorn and Elliott Tew, 2015, Kafai et al., 2019]. My thesis work began with a new theory of a programming skill, program tracing (starting in Section 3.1), then developed new ways to teach and assess. My personal experience doing this work is that theory is particularly useful for 1) imagining radically different research goals and solutions, and 2) reflecting upon designs for learning to then imagine improving them. I hope my work helps inspire others similarly. Earlier foundational work in CER also followed this pattern, from Mayer's work in the cognitivist tradition [Mayer, 1976, Mayer, 1981], to Papert's, Solomon's, and others' work in the constructionist tradition around LOGO [Solomon et al., 2020, Feurzeig et al., 1970].

Beyond theory use in general, my thesis work is an example of a specific type of theory development, starting more from computer science then drawing on existing education theory (versus starting from learning science and education theories brought to computing, as essential and important as that is). My theory work in Section 3.1 started from knowledge of computer science ideas such as symbolic execution of PL interpreter paths in newer program synthesis systems [Boyer et al., 1975, Torlak and Bodik, 2014]. My theory was also co-created as part of designing learning environments and assessments for program tracing skills, as a kind of design-based education research [Barab and Squire, 2004, Collins et al., 2004]. That combined approach has potential for new computing-specific knowledge and skill theories, as other sciences have progressed by adopting novel at the time computational representations; for example, cognitive science advancing psychology by using then novel computational representations made by Allen Newell and Herb Simon. For the cognitivist tradition of research new techniques in program synthesis and in AI should be taken as inspiration for new starting points for domain-specific theories of programming and computing knowledge. Computing education research is uniquely positioned to create theory starting from computing.

That kind of combined approach has happened before in computing education, creating the intelligent tutoring systems community from the LISP Tutor [Anderson et al., 1984]; ACT-R was the then-novel psychology, and production rule systems and probabilistic grammars were relatively novel computer science. At the same time we should learn from the missed opportunities of that work[3] for quickly and deeply engaging with social theories of learning as a part of design - see [Kelly et al., 2018,Deitrick et al., 2015,Deitrick et al., 2016] for design examples and [Engeström, 2015, Gutiérrez and Jurow, 2016] for theory.

This thesis and several lines of theory-based research build on a simple instructional idea: show all the knowledge represented in a theory of that knowledge; in CER this work has progressed for program writing but so much remains for program comprehension (much broader than program tracing). Work for learning program writing has drawn on worked examples [Morrison et al., 2015] and explicitly teaching programming patterns, with different theoretical framings as schemata, templates, and plans [Dalbey and Linn, 1985, Wiedenbeck, 1986, Spohrer et al., 1985]. While Schulte's program comprehension theory gives a basis for connecting ideas across different levels of comprehension [Schulte et al., 2010], we have had little to no work comparatively on teaching schemata, templates, and plans for program comprehension (or theoretical extensions of those constructs we may develop via such research). In particular, teaching the process of comprehension for specific purposes is underexplored; for example, the intersection between comprehension and writing: finding and reusing code (whereas many try to teach *writing* reusable code). I have only come across a few works here. One combined instruction and practice tool for higher-level program comprehension is built on the plan and beacon theory for program comprehension [Wiedenbeck, 1986], but only does a small part of that skill: beacon recognition [Leppan et al., 2007]. The other is LISP-KIE, a practice tool without instruction that scaffolds learners to learn code testing skills, which involves code comprehension skills, via a CodeProbe tool that provides a simplified interface for running code and getting its results, with the code of the program hidden or visible [Bell et al., 1994]. Some work on showing and sharing programming problem solving process may provide some of the tools foundation for future work in that large area [LaToza et al., 2020].

6.3.3 Assessment and theories of computing knowledge

Beyond this thesis as an example for using theory within design, this thesis also raises questions for theories of computing knowledge. Broadly speaking, theories of computing knowledge include, for example, "theories of what it means to know a programming language, what it means to know how to program, what it means to be an expert software engineer, what it means to have computer science literacy, and numerous other unanswered and yet foundational questions that are specific to computing education" [Nelson and Ko, 2018]. My work on differentiated assessment raises questions such as: For every advanced topic in computing, can a person know parts of it without knowing the prerequisites? Is it desirable to make questions that purely assess an advanced topic? For what topics is it possible to do that and why? How can we theoretically specify and separate

[3] And frankly of my own work, which has been quite cognitively focused on the whole, though with some design ideas broadly taken from social learning theories.

shallow, fragile ways learners may know these skills from more fluent and transferable knowledge?

6.3.3.1 Creating theories of knowledge via design-based research on assessment

Perhaps making assessment questions is a useful way of operationalizing theories of computing knowledge, as a kind of design-based research program [Barab and Squire, 2004, Collins et al., 2004]. For example, researchers might try to make more specific questions (as a 2017 ITiCSE WG did for programming fundamentals [Luxton-Reilly et al., 2018a]), try the assessments with actual learners, then iterate on their theory of computing knowledge, by perhaps questioning skills they cannot seem to assess, or adding skills inspired by an assessment they made.

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

The CER community might also want to analyze existing assessments made by teachers to develop theories of computing knowledge. This would draw on teachers' sense of what knowledge and skills are important to assess, which are expressed in question designs. This theory work must also avoid only using existing assessments which reify teachers' theories of the skill; observing expert practice of advanced skills in non-educational contexts is also critical.

6.3.3.2 Kane's validity framework: evaluating an assessment's immediate and broader consequences

My assessment design and validity work introduced Kane's assessment validity framework to computing education researchers. The key idea is to go beyond evaluating an assessment as a measurement, to considering the assessment's use and subsequent effects on learning outcomes — not just for the content of the test, but also more broadly such as self-efficacy, continued participation, equity, and identity. As computing education moves into education before college, our assessment work urgently needs to draw on Kane's framework. My differentiated assessment research with my collaborators analzyed the BDSI and argued most of the questions could not diagnose prerequisite skill issues; not surprising as the BDSI was not designed to do so. The BDSI was ground-breaking assessment work, but the research team was unaware of Kane's validity framework, and thus may not have considered designing to give formative feedback to learners. By using Kane's framework CER assessment researchers can start seeing those opportunities and take them. Without designing for positive effects on broad learning outcomes, it seems unlikely we will achieve them. For example, we should also measure the side-effects of our assessments on learners, as I did with self-efficacy (the first such evaluation I am aware of in CER).

6.4 Future Work

6.4.1 Assessment

With my precise formative assessment of program tracing, we can now systematically study existing instructional designs for basic PL knowledge, from the common presenting of a natural language definitions of PL constructs in textbooks and lectures, to particular design aspects of notional machines and practice for these skills. We can also compare language-specific assessments and language-independent ones, adding a new level of rigor to language-independent assessment work started with the FCS1 [Tew and Guzdial, 2011]. With this better and more comparable measure across studies, CER can improve the research process itself in that area.

To use the tracing assessments or develop new ones for other programming languages, please send an email to glnelson@uw.edu and I will happily share them with you. You can also recreate the test using the item design principles at the end of Section 4.2.3 and the item listing in Table 4.1.

The work on differentiated assessments contributed question designs and an analysis method for coding prerequisites, but requires empirical validity studies, as discussed in Section 5.7.6.1 (see Section 5.7.6 in general for other future work).

There are also broader areas for future work.

6.4.1.1 Program analysis to aid prerequisite analysis and barriers to full automation

To assign prerequisite tracing skills to an assessment question, one might use tools to analyze the code in the question. In earlier chapters, my theory of basic PL knowledge factors a PL interpreter into execution paths (see Section 3.1); a program analysis tool could detect the use of those in a question's code and then map them to prerequisite skill coding. A simpler manually annotated lexical mapping of code fragments in the interpreter to skills may work just as well in practice [4], especially for detecting the absence of a prerequisite. Symbolic execution of the question code might also be a flexible way to implement that analysis (especially for questions asking to reason about the code), but that also raises complexities - for example, while some code might execute or be used, is it actually required to answer the question?

To analyze the code in a question, one must also represent the task in the question prompt, which is not straightforward (and may require manual pre-processing of natural language question prompts, or natural language processing). For simple tracing tasks with given inputs this is straightforward, but less so for more complex advanced tasks like modifying code or answering questions about code properties. There may also be additional information in the question prompt that is not in the code itself; for example, a natural language question prompt may talk about some code being called concurrently, but the code itself has no concurrency controls that indicate that.

---

[4] Thanks Ras for this idea of trying a simpler solution.

6.4.1.2 *Mismatch between expert problem-solving procedure skill representations and novice's skills*

Beyond representing those tasks, there is the underlying problem of representing skills to model novice knowledge, rather than the highly factored and generalizable knowledge associated with expertise. While an expert's knowledge might be well modeled by an interpreter program, which will do programs *n* levels deep effortlessly (scoping, function calls, nesting etc), novices may need to develop fluency when using multiple skills at the same time. Thus, in my tracing assessment work I partly addressed this by modeling nested combinations. For example, one combination is a nested `if` statement where the outer `if` condition is true and the inner `if` condition is true; another separate combination covers the outer condition is false and inner condition is true. In the differentiated assessments coding of prerequisites, we used the collective wisdom of instructors and terms of art like "scoping" to group skills together. What is a general way to automate this? One close area of work is generating novice-learnable heuristic rules for problem solving, from an expert problem solving procedure [Butler, 2018]; those rules can also be seen as a better way to represent novice knowledge.

Partially automating the detection of higher level strategic skills is also promising but challenging, from representational skills like using variable tables to track values during tracing (to overcome human working memory limitations), to more difficult higher-level problem solving process and self-regulation skills.

6.4.2 *Program synthesis to generate curriculum, instruction, and assessment*

Starting from my model of PL knowledge as knowing nested combinations, future work can generate assessments and even a curriculum using program synthesis. Program synthesis techniques make the exponential search for desirable programs more feasible; for example, by bounding space of executions and restricting space of possible programs via sketches [Solar-lezama, 2008]. In education program synthesis has been used to generate problems that exercise a sequence of control flow paths through a solution procedure, but not a nested sequence of control flow paths [Andersen et al., 2013, Butler et al., 2015]. The synthesis can also be implemented to find programs that meet some of the curricular ordering principles from Section 3.2.2.4. By generating a curriculum in this way future work might automate the basic teaching of program tracing knowledge; careful evaluation of that curricula versus human-written curricula is important, and can use systematic assessments of that knowledge as I designed in Chapter 4.

This work is arguably more important than automating the detection of prerequisite and other skills, and modeling those skills. The underlying problem is how to design curricula and learning environments to

6.4.3 *Program comprehension learning tools*

How and to what extent do different program visualization designs and notional machine explanations help learning within self-contained learning tools like PLTutor? The current stack machine

visualization in PLTutor is probably sub-optimal due to visual spacing and eye travel alone, and also may present more details than are required (for instance, versus rewriting terms in the code or showing values next to variables and expressions). At the same time, the mechanicalness of the machine model may help scaffold causal inference. There are many potential studies for exploring this design choice, and the design choice in principle should be configurable by the learner, and also might change over time driven by an adaptive algorithm. Fundamentally my work also raises the question: what is the right granularity for stepping through those models, and highlighting and showing causal relationships within those designs? PLTutor's design is not the answer, just a starting point.

Another instructional design to experiment with is how to structure the program and machine execution state model to transfer to a mental tracing model. More studies asking learners to introspect and report what they are doing when mentally tracing may help this [Ericsson and Simon, 1993, Margulieux, 2019]. The mental sketchpad might resemble a physical paper one, with crossing off values and writing floating next to variables, as in this figure. Designers can observe people's practices with paper-based code tracing as a source of design ideas, such as putting finger of one hand on source of a function call to remember where to go back to. Studies of software developers in real environments, using various distributed cognition aids, would also be a rich source of ideas and practices.

Another larger design change for PLTutor is adding adaptivity; many generic designs for teaching a given knowledge model exist and could be adapted [Nesbit et al., 2015, Alkhatlan and Kalita, 2018, Crow et al., 2018]. One part will be adapting those systems to have a knowledge model with nested combinations (which can be handled perhaps by compiling it down to a graph representation). Adaptive teaching systems can use my theory of basic PL knowledge as the basis for new formal knowledge models to implement adaptive teaching. My theory also includes knowing the causal type of the computer (ontology) and functional forms of relationships, which are not usually assessed directly (like with direct conceptual questions or asking learners to highlight tokens in the code that cause it to have some behavior), instead indirectly by asking to perform program tracing skills. Teaching systems might thus correct and detect fundamental errors in causal type (i.e. root cause analysis for challenges in learning), and include those knowledge components as part of assessments and memory reinforcement practice like spaced repetition.

Beyond PLTutor-like tools, there is a large space for program comprehension tutorial systems teaching how to comprehend code, where program tracing is a particular strategy. For example, in practice developers use tools to help with debugging and tracing code, so the curriculum should be designed to teach for transfer, both to reach for and when using debugging and code comprehension tools. This might be done by integrating PLTutor directly into IDEs; for example, each PLTutor lesson would begin with the PLTutor agent opening some code in the IDE, then opening the debugger to step through it while giving instruction and assessments. Techniques from PLTutor and PStutor [Loksa, 2020] could help people learn to mix different mental tracing, code pattern/template recall, and debugging and code comprehension tools; work on programming process is closely related [LaToza et al., 2020] but not integrated into the specific tools for the skills, like demonstrating a step in-situ in a developer tool.

6.4.4 Inverting the curriculum and educational focus on writing code

My work started by questioning the assumption of focusing on writing when teaching computing; this suggests other underexplored areas for computing education and the culture of computing. When we teach we focus on writing solutions, especially from scratch; CER has invented new scaffolded writing problems, such as Parson's problems, where learners start from a scrambled correct solution and reorder it to work. Over time in K-12 and computational thinking CER has returned to small parts of Deimel, Moffat, and Kimura's ideas [Lee et al., 2011, Deimel and Naveda, 1990, Deimel and Makoid, 1985, Deimel and Moffat, 1982, Kimura, 1979] but slowly, incrementally, perhaps due to continued focus on writing. While a few recent works have returned to program comprehension learning [Izu et al., 2019, Busjahn and Schulte, 2013, Griffin, 2016, Griffin, 2018], CER should return to the richer and broader versions of those ideas in earlier works. To build on them and go beyond them, what would a broader comprehension curriculum like Deimel and Moffat's [Deimel and Moffat, 1982] look like in practice, designed with modern tooling?

What are radical ways we could invert the focus on writing in the curriculum? Applying program synthesis tools might come before writing programs from scratch, as strange as that might seem in our writing-focused teaching culture today; such tools could be used to modify existing programs then gradually move to writing. We also might teach refactoring and code modification tools early in the curriculum. We might incorporate more maintenance and even apprenticeship projects, with new tools for novices to comprehend codebases, supporting their learning journey and assessing their knowledge of the codebase and more generalizable skills. Such work can draw on professional tools and research on software engineering and documentation [Santos and Correia, 2020, Earle et al., 2015, Zhi et al., 2015], and bring a learning perspective to that area as well.

How might we foster a greater awareness of past solutions to problems, by reading and remembering great works of code? What might it mean to be "well read" in computing? Some expert programmers exhibit awareness like this; for example, Charles Simonyi speaks of studying an ALGOL compiler early in his learning and its lasting influence: "The Danish computer also had an incredible influence on me. At that time, it had probably the world's best Algol compiler, called Gier Algol. Before I went to Denmark, I had complete listings of the compiler, which I had studied inside and out...It was designed by Peter Naur. He is the letter N in BNF, the Backus Naur Form of syntax equations. I knew that program inside and out and I still know it...I think the Gier Algol program is still in my mind and influences my programming today. I always ask myself: if this were part of the Algol compiler, how would they do it? It's a very, very clever program." [Lammers, 1986] How do we teach that kind of awareness? What does teaching for transfer between comprehension and writing look like, from each direction?

Towards making this curriculum inclusive, instead of only teaching programs, we might teach the stories of people finding meaning in their programming activities, using old and contemporary programs written by people from diverse groups and backgrounds. How could it feel for a young girl to be introduced to loops by reading part of the Apollo moon lander's flight control computer written by Margaret Hamilton? As that young girl grows older she might revisit that code in a new ways, and hear stories around that code, how Margaret puzzled through solving particular problems with her team, while also balancing other parts of her life. Or, likewise, a young African American

girl reading programs by Katherine Johnson[5]? How might we teach within the true stories of how great works of computing were built, revealing the real and messy contexts, the meaningful lives and dreams of everyone involved, to see the technical as part of a greater whole?

Even if we make more inclusive curricula, a major challenge for learning computing is the lack of adults that know programming. For example, parents and grandparents can read to their children from storybooks, but there are few grandparents and platforms for reading "programming storybooks" like PLTutor with children. Perhaps we could invert that cultural practice to have kids make or customize a PLTtuor-like storybook[6], then bring it home and teach their parents and grandparents the basics of programming? This could be very empowering for learners and families alike (and other ways of learning together could be very empowering also [Banerjee et al., 2018]).

In closing, learning programming is hard, but maybe in part because our nascent computing culture focuses too much on theorizing programming as writing code mostly from scratch (particularly in education and higher education), instead of reading, building on, and reusing code. My work challenges those assumptions, presenting self-contained tools like PLTutor that can be deployed at initial stages of existing learning contexts and classes, and more systematic assessments that allow us to perceive and address on one potential cause for learner difficulties - weakness with basic PL knowledge. If we eliminate that basic and enduring barrier to programming, and put fluent comprehension in its place, could everyone in society learn to use programming as a tool and even as a medium [Kay, 2007]?

For example, let's take the foundational idea of using programming as a way of thinking, as a medium for new ideas and taking action in the world. Programming languages could show the underlying unity of concepts and ideas across domains, making them easier to learn. They could enable better ways to express, debate, and collaboratively improve social processes and institutions - advancing the very fabric of democratic society. Perhaps a key part of learning to think with programs and use them in those ways, is to learn to first remember, find, read, and build on the thinking of those that came before. Does that align with how we teach programming today? How do we make a culture where that is fun, meaningful, and valued, where expertise is writing as little as possible?

---

[5] Katherine was one of many pioneering African-American mathematicians and programmers at NASA in the 1960s.

[6] See my thesis defense talk to get a better idea of this (for the short version, start at 5:35 until I stop acting like a four year old, then 9:27 to 10:00).

BIBLIOGRAPHY

[Abrams, 1968] Abrams, M. D. (1968). A comparative sampling of the systems for producing computer-drawn flowcharts. In *Proceedings of the 1968 23rd ACM national conference* (pp. 743–750).

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

[Alexander et al.,] Alexander, C., Ishikawa, S., & Silverstein, M. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press.

[Alkhatlan and Kalita, 2018] Alkhatlan, A. & Kalita, J. (2018). Intelligent Tutoring Systems: A Comprehensive Historical Survey with Recent Developments http://arxiv.org/abs/1812.09628.

[Allen and Yen, 2001] Allen, M. J. & Yen, W. M. (2001). *Introduction to Measurement Theory*. Waveland Press.

[Almeida-Martínez et al., 2009] Almeida-Martínez, F. J., Urquiza-Fuentes, J., & Velázquez-Iturbide, J. (2009). Visualization of syntax trees for language processing courses. *J. Univers. Comput. Sci.*, *15*, 1546–1561.

[Andersen et al., 2013] Andersen, E., Gulwani, S., & Popovic, Z. (2013). A trace-based framework for analyzing and synthesizing educational progressions. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (pp. 773–782). http://delivery.acm.org/10.1145/2480000/2470764/p773-andersen.pdf?ip=147.46.43.52{&}id=2470764{&}acc=ACTIVESERVICE{&}key=0EC22F8658578FE1.D83A6478590749B7.4D4702B0C3E38B35.4D4702B0C3E38B35{&}CFID=359893123{&}CFTOKEN=38771235{&}{_}{_}acm{_}{_}=1403181674{_}a623ef71392610f696ef.

[Anderson et al., 1987] Anderson, J., Boyle, C., Farrell, R., & Reiser, B. (1987). Cognitive principles in the design of computer tutors http://act-r.psy.cmu.edu/wordpress/wp-content/uploads/2012/12/121CogPrinciples.pdf.

[Anderson et al., 1984] Anderson, J., Farrell, R., & Sauers, R. (1984). Learning to program in LISP. *Cognitive Science*, *8*(2), 87–129, https://doi.org/10.1016/S0364-0213(84)80013-0 http://doi.wiley.com/10.1016/S0364-0213(84)80013-0.

[Anderson et al., 1989] Anderson, J. R., Conrad, F. G., & Corbett, A. T. (1989). Skill acquisition and the LISP tutor. *Cognitive Science*, *13*(4), 467–505, https://doi.org/10.1016/0364-0213(89)90021-9.

[Arawjo et al., 2017] Arawjo, I., Wang, C.-y., Myers, A. C., Andersen, E., & Guimbretière, F. (2017). Teaching Programming with Gamified Semantics. In *Proceedings of the SIGCHI conference on Human factors in computing systems Reaching through technology - CHI '17* New York, New York, USA: ACM Press.

[Association for Computing Machinery, 2013] Association for Computing Machinery (2013). Computer science curricula 2013 https://www.acm.org/education/curricula-recommendations.

[Atkinson and Shiffrin, 1968] Atkinson, R. C. & Shiffrin, R. M. (1968). Human memory: A proposed system and its control processes. In *Psychology of learning and motivation*, volume 2 (pp. 89–195). Elsevier.

[Atkinson et al., 2000] Atkinson, R. K., Derry, S. J., Renkl, A., & Wortham, D. (2000). Learning from examples: Instructional principles from the worked examples research. *Review of Educational Research*, *70*(2), 181–214.

[Baecker, 1975] Baecker, R. (1975). Two systems which produce animated representations of the execution of computer programs. In *Proceedings of the Fifth SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '75 (pp. 158–167). New York, NY, USA: Association for Computing Machinery https://doi.org/10.1145/800284.811152.

[Baecker, 1998] Baecker, R. (1998). The Early History of Software Visualization. In J. T. Stasko, M. H. Brown, J. B. Domingue, & B. A. Price (Eds.), *Software Visualization*. MIT Press http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=43C8D1754035F2E9A4509F69BDB29CC2?doi=10.1.1.178.9799&rep=rep1&type=pdf.

[Bailes et al., 2006] Bailes, S., Libman, E., Baltzan, M., Amsel, R., Schondorf, R., & Fichten, C. S. (2006). Brief and distinct empirical sleepiness and fatigue scales. *Journal of Psychosomatic Research*, *60*(6), 605–613, https://doi.org/10.1016/j.jpsychores.2005.08.015.

[Banerjee et al., 2018] Banerjee, R., et al. (2018). Empowering families facing English literacy challenges to jointly engage in computer programming. In *Conference on Human Factors in Computing Systems - Proceedings*, volume 2018-April https://doi.org/10.1145/3173574.3174196.

[Barab and Squire, 2004] Barab, S. & Squire, K. (2004). Design-based research: Putting a stake in the ground. *Journal of the Learning Sciences*, *13*(1), 1–14, https://doi.org/10.1207/s15327809jls1301_1 https://doi.org/10.1207/s15327809jls1301_1.

[Barr et al., 1975a] Barr, A., Beard, M., & Atkinson, R. C. (1975a). A rationale and description of a CAI program to teach the BASIC programming language. *Instructional Science*, *4*(1), 1–31, https://doi.org/10.1007/BF00157068 http://link.springer.com/10.1007/BF00157068.

[Barr et al., 1975b] Barr, A., Beard, M., & Atkinson, R. C. (1975b). *THE COMPUTER AS A TUTORIAL LABORATORY: THE STANFORD BIP PROJECT Technical Report 260*. Technical report, STANFORD UNIV CA INST FOR MATHEMATICAL STUDIES IN THE SOCIAL SCIENCES, OFFICE OF NAVAL RESEARCH.

[Bauer et al., 2018] Bauer, A., O'Rourke, E., Thayer, K., Butler, E., Brand, W., & Reges, S. (2018). *Practicum: a scalable online system for faded worked examples in CS1 UW-CSE-18-09-01*. Technical report https://cs.carleton.edu/faculty/awb/data/Bauer_Practicum_2018.pdf.

[Bayman and Mayer, 1988] Bayman, P. & Mayer, R. E. (1988). Using conceptual models to teach BASIC computer programming. *Journal of Educational Psychology*, *80*(3), 291–298, https://doi.org/10.1037/0022-0663.80.3.291.

[Bell et al., 1994] Bell, J. E., Linn, M. C., & Clancy, M. (1994). Knowledge Integration in Introductory Programming: CodeProbe and Interactive Case Studies. *Interactive Learning Environments*, *4*(1), 75–95, https://doi.org/10.1080/1049482940040103.

[Ben-Ari, 2001] Ben-Ari, M. (2001). Constructivism in computer science education. *Journal of Computers in Mathematics and Science Teaching*, *20*(1), 45–73.

[Bergin,] Bergin, J. Pedagogical Patterns https://web.archive.org/web/20020606104041/http://sol.info.unlp.edu.ar/ppp/learn.htm.

[Bergin, 2000] Bergin, J. (2000). Fourteen pedagogical patterns. In *Proceedings of the Fifth European Conference on Pattern Languages of Programs (EuroPLop 2000)* (pp. 1–39). Irsee, Germany http://hillside.net/europlop/HillsideEurope/Papers/EuroPLoP2000/2000_Bergin_FourteenPedagogicalPatterns.pdf.

[Berry, 1991] Berry, D. (1991). *Generating Program Animators from Programming Language Semantics*. PhD thesis, University of Edinburgh.

[Bimba et al., 2017] Bimba, A. T., Idris, N., Mahmud, R. B., & Al-Hunaiyyan, A. A. (2017). A cognitive knowledge-based framework for adaptive feedback. *Advances in Intelligent Systems and Computing*, *532*, 245–255, https://doi.org/10.1007/978-3-319-48517-1_22.

[Blackwell et al., 2007] Blackwell, L. S., Trzesniewski, K. H., & Dweck, C. S. (2007). Implicit theories of intelligence predict achievement across an adolescent transition: a longitudinal study and an intervention. *Child development*, *78*(1), 246–63, https://doi.org/10.1111/j.1467-8624.2007.00995.x http://www.ncbi.nlm.nih.gov/pubmed/17328703.

[Bonate, 2000] Bonate, P. L. (2000). *Analysis of Pretest-Posttest Designs*. CRC Press.

[Bork, 1971] Bork, A. M. (1971). Learning to program for the science student. *Journal ofEducational Data Processing*, *8*(5), 1–5 https://files.eric.ed.gov/fulltext/ED060627.pdf.

[Borsboom and Mellenbergh, 2007] Borsboom, D. & Mellenbergh, G. J. (2007). Test validity in cognitive assessment. In J. Leighton & M. Gierl (Eds.), *Cognitive Diagnostic Assessment for Education: Theory and Applications* (pp. 85–116). Cambridge: Cambridge University Press https://www.cambridge.org/core/product/identifier/CBO9780511611186A011/type/book{_}part.

[Boyer et al., 1975] Boyer, R. S., Elspas, B., & Levitt, K. N. (1975). Select—a formal system for testing and debugging programs by symbolic execution. *SIGPLAN Not.*, *10*(6), 234–245, https://doi.org/10.1145/390016.808445 https://doi.org/10.1145/390016.808445.

[Bransford et al., 2000] Bransford, J. D., Brown, A. L., & Cocking, R. R. (2000). *How People Learn: Brain, Mind, Experience, and School: Expanded Edition*. National Academies Press http://www.nap.edu/catalog/9853.html.

[Brooks, 1983] Brooks, R. (1983). Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, *18*(6), 543–554, https://doi.org/10.1016/S0020-7373(83)80031-5 http://www.scopus.com/inward/record.url?eid=2-s2.0-0020766507{&}partnerID=tZOtx3y1{%}5Cnhttp://linkinghub.elsevier.com/retrieve/pii/S0020737383800315http://linkinghub.elsevier.com/retrieve/pii/S0020737383800315.

[Brooks, 1999] Brooks, R. (1999). Towards a theory of the cognitive processes in computer programming. *International Journal of Human-Computer Studies*, *51*(2), 197–211, https://doi.org/10.1006/ijhc.1977.0306 http://dx.doi.org/10.1006/ijhc.1977.0306http://linkinghub.elsevier.com/retrieve/pii/S1071581977603062.

[Brown et al., 2001] Brown, L., Cai, T., & DasGupta, A. (2001). Interval estimation for a binomial proportion. *Statistical Science*, *16*(2), 101–133. cited By 988 https://www.scopus.com/inward/record.uri?eid=2-s2.0-0000460102&partnerID=40&md5=0997011d7da77720486e29c728a95d34.

[Bruce-Lockhart et al., 2009] Bruce-Lockhart, M., Crescenzi, P., & Norvell, T. (2009). Integrating test generation functionality into the Teaching Machine environment. *Electronic Notes in Theoretical Computer Science*, *224*(C), 115–124, https://doi.org/10.1016/j.entcs.2008.12.055 http://dx.doi.org/10.1016/j.entcs.2008.12.055.

[Brusilovsky, 1992] Brusilovsky, P. (1992). Intelligent Tutor, Environment and Manual for Introductory Programming. *Educational and Training Technology International*, *29*(1), 26–34, https://doi.org/10.1080/0954730920290104 https://www.tandfonline.com/doi/full/10.1080/0954730920290104.

[Brusilovsky, 1994] Brusilovsky, P. (1994). Explanatory visualization in an educational programming environment: Connecting examples with general knowledge. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *876 LNCS*, 202–212, https://doi.org/10.1007/3-540-58648-2_38.

[Brusilovsky, 2004] Brusilovsky, P. (2004). Knowledgetree: A distributed architecture for adaptive e-learning. In *Proceedings of the 13th International World Wide Web Conference on Alternate Track Papers amp; Posters*, WWW Alt. '04 (pp. 104–113). New York, NY, USA: Association for Computing Machinery https://doi.org/10.1145/1013367.1013386.

[Brusilovsky and Loboda, 2006] Brusilovsky, P. & Loboda, T. (2006). WADEIn II: a case for adaptive explanatory visualization. *ACM SIGCSE Bulletin*, (pp. 48–52)., https://doi.org/10.1145/1140124.1140140 http://dl.acm.org/citation.cfm?id=1140140.

[Brusilovsky et al., 2015] Brusilovsky, P., et al. (2015). Increasing Adoption of Smart Learning Content for Computer Science Education. (March 2015), 31–57, https://doi.org/10.1145/2713609.2713611.

[Brusilovsky and Su, 2002] Brusilovsky, P. & Su, H.-D. (2002). Adaptive Visualization Component of a Distributed Web-Based Adaptive Educational System. *International Conference on Intelligent Tutoring Systems*, (pp. 229–238)., https://doi.org/10.1007/3-540-47987-2_27.

[Burnett et al., 2016] Burnett, M., et al. (2016). GenderMag: A method for evaluating software's gender inclusiveness. *Interacting with Computers*, *28*(6), 760–787, https://doi.org/10.1093/iwc/iwv046.

[Busjahn and Schulte, 2013] Busjahn, T. & Schulte, C. (2013). The use of code reading in teaching programming. *Proceedings of the 13th Koli Calling International Conference on Computing Education Research*, (pp. 3–11)., https://doi.org/10.1145/2526968.2526969 http://doi.acm.org/10.1145/2526968.2526969.

[Butler et al., 2015] Butler, E., Andersen, E., Smith, A. M., Gulwani, S., & Popović, Z. (2015). Automatic Game Progression Design through Analysis of Solution Features. *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems - CHI '15*, (pp. 2407–2416)., https://doi.org/10.1145/2702123.2702330 http://dl.acm.org/citation.cfm?doid=2702123.2702330.

[Butler, 2018] Butler, E. D. (2018). *Automatic Generation of Procedural Knowledge Using Program Synthesis*. PhD thesis, University of Washington http://hdl.handle.net/1773/43656.

[Caceffo et al., 2018] Caceffo, R., Gama, G., Benatti, R., Aparecida, T., Caldas, T., & Azevedo, R. (2018). *A Concept Inventory for CS1 Introductory Programming Courses in C*. Technical report, University of Campinas, SP, Brasil.

[Caceffo et al., 2016] Caceffo, R., Wolfman, S., Booth, K. S., & Azevedo, R. (2016). Developing a computer science concept inventory for introductory programming. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, SIGCSE '16 (pp. 364–369). New York, NY, USA: Association for Computing Machinery https://doi.org/10.1145/2839509.2844559.

[Campbell and Bolker, 2002] Campbell, W. & Bolker, E. (2002). Teaching programming by immersion, reading and writing. In *32nd Annual Frontiers in Education*, volume 1 (pp. T4G–23–T4G–28).: IEEE http://ieeexplore.ieee.org/document/1158015/.

[Castro-Schez et al., 2020] Castro-Schez, J. J., González-Morcillo, C., Albusac, J., & Vallejo-Fernandez, D. (2020). An intelligent tutoring system for supporting active learning: A case study on predictive parsing learning. *Information Sciences*, *544*, 446 – 468.

[Clark et al., 1998] Clark, D., MacNish, C., & Royle, G. F. (1998). Java as a teaching language—opportunities, pitfalls and solutions. *Proceedings of the 3rd Australasian conference on Computer science education*, (pp. 173–179)., https://doi.org/10.1145/289393.289418.

[Clear et al., 2011] Clear, T., Whalley, J., Robbins, P., Philpott, A., Eckerdal, A., & Laakso, M. (2011). Report on the final bracelet workshop: Auckland university of technology, september 2010. *Journal of Applied Computing and Information Technology*, *15* http://aut.researchgateway.ac.nz/handle/10292/1514.

[Clements, 2006] Clements, J. (2006). *Portable and high-level access to the stack with Continuation Marks*. PhD thesis, Northeastern University.

[Clements et al., 2001] Clements, J., Flatt, M., & Felleisen, M. (2001). Modeling an Algebraic Stepper. In *European Symposium on Programming* (pp. 320–334). http://link.springer.com/10.1007/3-540-45309-1_21.

[Codecademy, 2016] Codecademy (2016). https://www.codecademy.com. Accessed: 2016-12-12.

[CodingBat, 2016] CodingBat (2016). https://www.codingbat.com. Accessed: 2016-12-12.

[Collins et al., 2004] Collins, A., Joseph, D., & Bielaczyc, K. (2004). Design research: Theoretical and methodological issues. *Journal of the Learning Sciences*, *13*(1), 15–42, https://doi.org/10.1207/s15327809jls1301_2 https://doi.org/10.1207/s15327809jls1301_2.

[Corney et al., 2014] Corney, M., Fitzgerald, S., Hanks, B., Lister, R., McCauley, R., & Murphy, L. (2014). "explain in plain english" questions revisited: Data structures problems. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, SIGCSE '14 (pp. 591–596). New York, NY, USA: Association for Computing Machinery https://doi.org/10.1145/2538862.2538911.

[Council, 2012] Council, N. R. (2012). *Discipline-Based Education Research: Understanding and Improving Learning in Undergraduate Science and Engineering*. Washington, DC: The National Academies Press https://www.nap.edu/catalog/13362/discipline-based-education-research-understanding-and-improving-learning-in-undergr

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

[Cross II et al., 2002] Cross II, J. H., Hendrix, T. D., & Barowski, L. A. (2002). Using the debugger as an integral part of teaching cs1. In *32nd Annual Frontiers in Education*, volume 2 (pp. F1G–F1G). Chapaign, IL, USA: IEEE Stripes Publishing LLC.

[Crow et al., 2018] Crow, T., Luxton-Reilly, A., & Wuensche, B. (2018). Intelligent tutoring systems for programming education: A systematic review. In *Proceedings of the 20th Australasian Computing Education Conference*, ACE '18 (pp. 53–62). New York, NY, USA: Association for Computing Machinery https://doi.org/10.1145/3160489.3160492.

[Cunningham et al., 2017] Cunningham, K., Blanchard, S., Ericson, B., & Guzdial, M. (2017). Using Tracing and Sketching to Solve Programming Problems. In *ICER '17* (pp. 164–172).: ACM Press http://dl.acm.org/citation.cfm?doid=3105726.3106190.

[Curcio et al., 2006] Curcio, G., Ferrara, M., & De Gennaro, L. (2006). Sleep loss, learning capacity and academic performance. *Sleep Medicine Reviews*, *10*(5), 323–337, https://doi.org/10.1016/j.smrv.2005.11.001.

[Cutts et al., 2019] Cutts, Q., et al. (2019). Experience report: Thinkathon – countering an "i got it working" mentality with pencil-and-paper exercises. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '19 (pp.

203–209). New York, NY, USA: Association for Computing Machinery https://doi.org/10.1145/3304221.3319785.

[Dalbey and Linn, 1985] Dalbey, J. & Linn, M. C. (1985). The demands and requirements of computer programming: A review of the literature. *Journal of Educational Computing Research*, *1*(3), 253–274, https://doi.org/10.2190/BC76-8479-YM0X-7FUA.

[Danielsiek et al., 2017] Danielsiek, H., Toma, L., & Vahrenhold, J. (2017). An Instrument to Assess Self-Efficacy in Introductory Algorithms Courses. In *ICER* (pp. 217–225). New York, New York, USA: ACM Press http://dl.acm.org/citation.cfm?doid=3105726.3106171.

[Danielson and Nievergelt, 1975] Danielson, R. L. & Nievergelt, J. (1975). An automatic tutor for introductory programming students. *ACM SIGCSE Bulletin*, *7*(1), 47–50, https://doi.org/10.1145/953064.811129.

[de Jong, 2010] de Jong, T. (2010). Cognitive load theory, educational research, and instructional design: Some food for thought. *Instructional Science*, *38*(2), 105–134.

[De La Torre, 2011] De La Torre, J. (2011). The generalized DINA model framework. *Psychometrika*, *76*(2), 179–199.

[De La Torre et al., 2010] De La Torre, J., Hong, Y., & Deng, W. (2010). Factors affecting the item parameter estimation and classification accuracy of the dina model. *Journal of Educational Measurement*, *47*(2), 227–249, https://doi.org/10.1111/j.1745-3984.2010.00110.x https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1745-3984.2010.00110.x.

[de la Torre and Minchen, 2014] de la Torre, J. & Minchen, N. (2014). Cognitively Diagnostic Assessments and the Cognitive Diagnosis Model Framework. *Psicología Educativa*, *20*(2), 89–97, https://doi.org/10.1016/j.pse.2014.11.001 www.pearsoneducacion.net/woolfolkhttp://journals.copmadrid.org/psed/articulo.php?id=3e15cc11f979ed25912dff5b0669f2cd.

[De Raadt et al., 2002] De Raadt, M., Toleman, M., & Watson, R. (2002). Language Trends in Introductory Programming Courses On the internet. *Informing Science*, (pp. 329–337). http://proceedings.informingscience.org/IS2002Proceedings/papers/deRaa136Langu.pdf.

[Decker, 2007] Decker, A. (2007). *HOW STUDENTS MEASURE UP: AN ASSESSMENT INSTRUMENT FOR INTRODUCTORY COMPUTER SCIENCE*. PhD thesis.

[Decker and McGill, 2019] Decker, A. & McGill, M. M. (2019). A Topical Review of Evaluation Instruments for Computing Education. In *SIGCSE* (pp. 558–564).: ACM Press http://dl.acm.org/citation.cfm?doid=3287324.3287393.

[Deimel and Moffat, 1982] Deimel, L. & Moffat, D. (1982). A More Analytical Approach to Teaching the Introductory Programming Course. In *Proceedings of the National Educational Computing Conference* (pp. 114–118).

[Deimel and Naveda, 1990] Deimel, L. E. & Naveda, J. F. (1990). *Reading Computer Programs: Instructor's Guide to Exercises*. Number CMU/SEI-90-EM-3 http://www.dtic.mil/docs/citations/ADA228026.

[Deimel and Makoid, 1985] Deimel, L. E. J. & Makoid, L. (1985). Developing program reading comprehension tests for the Computer Science classroom. *Proc. IFIP TC 34th World Conf. on Computers in Education WCEE 85*, (pp. 535–540).

[Deitrick et al., 2015] Deitrick, E., Shapiro, R. B., Ahrens, M. P., Fiebrink, R., Lehrman, P. D., & Farooq, S. (2015). Using distributed cognition theory to analyze collaborative computer science learning. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research*, ICER '15 (pp. 51–60). New York, NY, USA: Association for Computing Machinery https://doi.org/10.1145/2787622.2787715.

[Deitrick et al., 2016] Deitrick, E., Shapiro, R. B., & Gravel, B. (2016). How do we assess equity in programming pairs? Singapore: International Society of the Learning Sciences.

[Denick et al., 2012] Denick, D., Santiago-Román, A., Streveler, R., & Barrett, N. (2012). Validating of the diagnostic capabilities of concept inventories: Preliminary evidence from the Concept Assessment Tool for Statics (CATS). In *2012 ASEE Annual Conference & Exposition Proceedings* (pp. 25.1457.1–25.1457.19). San Antonio, Texas: ASEE Conferences http://peer.asee.org/22214.

[Dewald et al., 2010] Dewald, J. F., Meijer, A. M., Oort, F. J., Kerkhof, G. A., & Bögels, S. M. (2010). The influence of sleep quality, sleep duration and sleepiness on school performance in children and adolescents: A meta-analytic review. *Sleep Medicine Reviews*, *14*(3), 179–189, https://doi.org/10.1016/j.smrv.2009.10.004 http://dx.doi.org/10.1016/j.smrv.2009.10.004.

[Dorn and Elliott Tew, 2015] Dorn, B. & Elliott Tew, A. (2015). Empirical validation and application of the computing attitudes survey. *Computer Science Education*, *25*(1), 1–36, https://doi.org/10.1080/08993408.2015.1014142 http://dx.doi.org/10.1080/08993408.2015.1014142http://www.tandfonline.com/doi/full/10.1080/08993408.2015.1014142.

[Downey and Stein, 2006] Downey, A. & Stein, L. (2006). Designing a small-footprint curriculum in computer science. In *Proceedings. Frontiers in Education. 36th Annual Conference* (pp. 21–26).: IEEE http://ieeexplore.ieee.org/document/4117160/.

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

[du Boulay and O'Shea, 1976] du Boulay, B. & O'Shea, T. (1976). *How to work the LOGO machine: a primer for ELOGO*. Technical report.

[du Boulay et al., 1981] du Boulay, B., O'Shea, T., & Monk, J. (1981). The black box inside the glass box: presenting computing concepts to novices. *International Journal of Man-Machine Studies*, *14*(3), 237–249, https://doi.org/10.1016/S0020-7373(81)80056-9 http://linkinghub.elsevier.com/retrieve/pii/S1071581981603092http://linkinghub.elsevier.com/retrieve/pii/S0020737381800569.

[Dunn and Mulvenon, 2009] Dunn, K. E. & Mulvenon, S. W. (2009). A Critical Review of Research on Formative Assessment: The Limited Scientific Evidence of the Impact of Formative Assessment in Education. *Practical Assessment, Research & Evaluation*, *14*(7), https://doi.org/10.1002/ir https://pareonline.net/pdf/v14n7.pdf.

[Duran et al., 2019] Duran, R., Rybicki, J.-M., Sorva, J., & Hellas, A. (2019). Exploring the Value of Student Self-Evaluation in Introductory Programming. In *ICER '19* (pp. 121–130).: ACM Press http://dl.acm.org/citation.cfm?doid=3291279.3339407.

[Dyck and Mayer, 1989] Dyck, J. L. & Mayer, R. E. (1989). Teaching for Transfer of Computer Program Comprehension Skill. *Journal of Educational Psychology*, *81*(1), 16–24, https://doi.org/10.1037//0022-0663.81.1.16.

[Earle et al., 2015] Earle, R. H., Rosso, M. A., & Alexander, K. E. (2015). User preferences of software documentation genres. *SIGDOC 2015 - Proceedings of the 33rd Annual International Conference on the Design of Communication*, https://doi.org/10.1145/2775441.2775457.

[Eisenstadt et al., 1993] Eisenstadt, M., Price, B. A., & Domingue, J. (1993). Redressing ITS Fallacies Via Software Visualization. In *Cognitive Models and Intelligent Environments for Learning Programming* (pp. 220–234). http://link.springer.com/10.1007/978-3-662-11334-9_20.

[Elliott Tew, 2010] Elliott Tew, A. (2010). *Assessing fundamental introductory computing concept knowledge in a language independent manner*. PhD thesis

http://search.proquest.com/docview/873212789?accountid=14696{%}5Cnhttp://sfx.umd.edu/cp?url{_}ver=Z39.88-2004{&}rft{_}val{_}fmt=info:ofi/fmt:kev:mtx:dissertation{&}genre=dissertations+{%}26+theses{&}sid=ProQ:ProQuest+Dissertations+{%}26+Theses+Full+Text{&}atitle={&}title=Asses.

[Engeström, 2015] Engeström, Y. (2015). *Learning by expanding*. Cambridge University Press.

[Ericson et al., 2018] Ericson, B. J., Foley, J. D., & Rick, J. (2018). Evaluating the Efficiency and Effectiveness of Adaptive Parsons Problems. In *ICER '18* (pp. 60–68).: ACM Press http://dl.acm.org/citation.cfm?doid=3230977.3231000.

[Ericsson and Simon, 1993] Ericsson, K. A. & Simon, H. A. (1993). *Protocol Analysis: Verbal Reports as Data Revised Edition*. The MIT Press.

[Evans and Darley, 1966] Evans, T. G. & Darley, D. L. (1966). On-line debugging techniques: A survey. In *Proceedings of the November 7-10, 1966, Fall Joint Computer Conference*, AFIPS '66 (Fall) (pp. 37–50). New York, NY, USA: Association for Computing Machinery https://doi.org/10.1145/1464291.1464295.

[Farrell et al., 1984] Farrell, R. G., Anderson, J. R., & Reiser, B. J. (1984). An Interactive Computer Based Tutor for {LISP}. In *AAAI84* (pp. 106–109).

[Felleisen et al., 2001] Felleisen, M., Findler, R. B., Flatt, M., & Krishnamurthi, S. (2001). How to Design Programs. *MIT Press*, (pp. 720)., https://doi.org/10.1136/bjsm.27.1.58.

[Feurzeig et al., 1970] Feurzeig, W., Papert, S., Bloom, M., Grant, R., & Solomon, C. (1970). Programming-languages as a conceptual framework for teaching mathematics. *ACM SIGCUE Outlook*, *4*(2), 13–17.

[Fincher et al., 2020] Fincher, S., et al. (2020). Notional machines in computing education: The education of attention. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education*, ITiCSE-WGR '20 (pp. 21–50). New York, NY, USA: Association for Computing Machinery https://doi.org/10.1145/3437800.3439202.

[Fincher and Robins, 2019] Fincher, S. A. & Robins, A. V. (2019). Pedagogic approaches. In *The Cambridge handbook of computing education research*. Cambridge University Press.

[Fisler et al., 2017] Fisler, K., Krishnamurthi, S., & Tunnell Wilson, P. (2017). Assessing and teaching scope, mutation, and aliasing in upper-level undergraduates. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '17 (pp. 213–218). New York, NY, USA: Association for Computing Machinery https://doi.org/10.1145/3017680.3017777.

[Fortunato, 2017] Fortunato, M. W. (2017). Advancing educational diversity: antifragility, standardization, democracy, and a multitude of education options. *Cultural Studies of Science Education*, *12*(1), 177–187, https://doi.org/10.1007/s11422-016-9754-4.

[Fouh et al., 2014] Fouh, E., Karavirta, V., Breakiron, D., Hamouda, S., Hall, S., Naps, T., & Shaffer, C. (2014). Design and architecture of an interactive etextbook the opendsa system. *Science of Computer Programming*, *88*(1), 22–40, https://doi.org/10.1016/j.scico.2013.11.040.

[Friend, 1973] Friend, J. (1973). *Computer-Assisted Instruction in Programming: A Curriculum Description Technical Report Number 211*. Technical report, Stanford UniV., Calif. Inst. for Mathematical Studies in Social Science., https://web.stanford.edu/group/csli-suppes/techreports/IMSSS_211.pdf.

[Gajraj et al., 2011] Gajraj, R. R., Williams, M., Bernard, M., & Singh, L. (2011). Transforming Source Code Examples into Programming Tutorials. *The Sixth International Multi-Conference on Computing in the Global Information Technology*, (May 2014), 160–164.

[Galeshi, 2012] Galeshi, R. (2012). *Cognitive Diagnostic Model, a Simulated-Based Study: Understanding Compensatory Reparameterized Unified Model (C-RUM)*. PhD thesis, Virginia Polytechnic Institute and State University http://hdl.handle.net/10919/77228.

[Giordano et al., 2015] Giordano, D., Maiorana, F., Csizmadia, A. P., Marsden, S., Riedesel, C., Mishra, S., & Vinikienundefined, L. (2015). New horizons in the assessment of computer science at school and beyond: Leveraging on the viva platform. In *Proceedings of the 2015 ITiCSE on Working Group Reports*, ITICSE-WGR '15 (pp. 117–147). New York, NY, USA: Association for Computing Machinery https://doi.org/10.1145/2858796.2858801.

[Gluga et al., 2012] Gluga, R., Kay, J., Lister, R., Kleitman, S., & Lever, T. (2012). Coming to terms with Bloom : an online tutorial for teachers of programming fundamentals. In *ACE '12: Proceedings of the Fourteenth Australasian Computing Education Conference - Volume 123* (pp. 147–156). AUS: Australian Computer Society, Inc.

[Goldberg and Kay, 1976] Goldberg, A. & Kay, A. (1976). *SMALLTALK-72 INSTRUCTION MANUAL*. Xerox Corporation.

[Goldman et al., 2008] Goldman, K., Gross, P., Heeren, C., Herman, G., Kaczmarczyk, L., Loui, M. C., & Zilles, C. (2008). Identifying important and difficult concepts in introductory computing courses using a delphi process. *SIGCSE Bull.*, *40*(1), 256–260, https://doi.org/10.1145/1352322.1352226 https://doi.org/10.1145/1352322.1352226.

[Goldman et al., 2010] Goldman, K., Gross, P., Heeren, C., Herman, G. L., Kaczmarczyk, L., Loui, M. C., & Zilles, C. (2010). Setting the scope of concept inventories for introductory computing subjects. *ACM Trans. Comput. Educ.*, *10*(2), https://doi.org/10.1145/1789934.1789935 https://doi.org/10.1145/1789934.1789935.

[Goldstein, 1979] Goldstein, I. P. (1979). The genetic graph: a representation for the evolution of procedural knowledge. *International Journal of Man-Machine Studies*, *11*(1), 51–77, https://doi.org/10.1016/S0020-7373(79)80005-X.

[Golemanov and Golemanova, 2020] Golemanov, T. & Golemanova, E. (2020). A set of tools to teach language processors construction. *Proceedings of the 21st International Conference on Computer Systems and Technologies '20*.

[Griffin, 2016] Griffin, J. M. (2016). Learning by taking apart: deconstructing code by reading, tracing, and debugging. In *Conference on Information Technology Education* (pp. 148–153).: ACM.

[Griffin, 2018] Griffin, J. M. (2018). *LEARNING TO PROGRAM FROM INTERACTIVE EXAMPLE CODE (WITH AND WITHOUT INTENTIONAL BUGS)*. PhD thesis, Temple University.

[Griffiths and Tenenbaum, 2009] Griffiths, T. L. & Tenenbaum, J. B. (2009). Theory-based causal induction. *Psychological Review*, *116*(4), 661–716, https://doi.org/10.1037/a0017201 http://doi.apa.org/getdoi.cfm?doi=10.1037/a0017201.

[Guo, 2013a] Guo, P. J. (2013a). Online python tutor. In *Proceeding of the 44th ACM technical symposium on Computer science education - SIGCSE '13* (pp. 579). New York, New York, USA: ACM Press http://dl.acm.org/citation.cfm?doid=2445196.2445368.

[Guo, 2013b] Guo, P. J. (2013b). Online python tutor: Embeddable web-based program visualization for cs education. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13 (pp. 579–584). New York, NY, USA: Association for Computing Machinery https://doi.org/10.1145/2445196.2445368.

[Gutiérrez and Jurow, 2016] Gutiérrez, K. D. & Jurow, A. S. (2016). Social Design Experiments: Toward Equity by Design. *Journal of the Learning Sciences*, *25*(4), 565–598, https://doi.org/10.1080/10508406.2016.1204548.

[Guzdial et al., 2019] Guzdial, M., Krishnamurthi, S., Sorva, J., & Vahrenhold, J. (2019). Notional Machines and Programming Language Semantics in Education (Dagstuhl Seminar 19281). *Dagstuhl Reports*, *9*(7), 1–23, https://doi.org/10.4230/DagRep.9.7.1 https://drops.dagstuhl.de/opus/volltexte/2019/11627.

[Hamouda et al., 2017] Hamouda, S., Edwards, S. H., Elmongui, H. G., Ernst, J. V., & Shaffer, C. A. (2017). A basic recursion concept inventory. *Computer Science Education*, *27*(2), 121–148, https://doi.org/10.1080/08993408.2017.1414728 https://www.tandfonline.com/action/journalInformation?journalCode=ncse20.

[Harrington and Cheng, 2018] Harrington, B. & Cheng, N. (2018). Tracing vs. writing code: Beyond the learning hierarchy. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, SIGCSE '18 (pp. 423–428). New York, NY, USA: Association for Computing Machinery https://doi.org/10.1145/3159450.3159530.

[Hattie, 1999] Hattie, J. (1999). *Influences on student Learning*. Technical report https://cdn.auckland.ac.nz/assets/education/about/research/documents/influences-on-student-learning.pdf.

[Hattie, 2008] Hattie, J. (2008). *Visible learning: A synthesis of over 800 meta-analyses relating to achievement*. routledge https://apprendre.auf.org/wp-content/opera/13-BF-References-et-biblio-RPT-2014/VisibleLearning_AsynthesisorOver800Meta-analysesRelatingtoAchievement_HattieJ2009...pdf.

[Hattie and Timperley, 2007] Hattie, J. & Timperley, H. (2007). The Power of Feedback. *Review of Educational Research*, *77*(1), 81–112, https://doi.org/10.3102/003465430298487 http://journals.sagepub.com/doi/10.3102/003465430298487.

[Hendrix et al., 2007] Hendrix, D., Cross, J. H., Jain, J., & Barowski, L. (2007). Providing Data Structure Animations in a Lightweight IDE. *Electronic Notes in Theoretical Computer Science*, *178*, 101–109, https://doi.org/10.1016/j.entcs.2007.01.039.

[Hertz and Jump, 2013] Hertz, M. & Jump, M. (2013). Trace-Based Teaching in Early Programming Courses. *Proceedings of the 44th ACM Technical Symposium on Computer Science Education*, (pp. 561–566)., https://doi.org/10.1145/2445196.2445364 http://dl.acm.org/citation.cfm?doid=2445196.2445364.

[Hestenes et al., 1992] Hestenes, D., Wells, M., & Swackhamer, G. (1992). Force concept inventory. *The physics teacher*, *30*(3), 141–158.

[Hidalgo-Céspedes et al., 2016] Hidalgo-Céspedes, J., Marín-Raventós, G., & Lara-Villagrán, V. (2016). Learning principles in program visualizations: A systematic literature review. In *Proceedings - Frontiers in Education Conference, FIE*, volume 2016-Novem.

[Hinkelmann and Kempthorne, 2008] Hinkelmann, K. & Kempthorne, O. (2008). *Design and Analysis of Experiments, Volume I: Introduction to Experimental Design*. Wiley.

[Hoc and Nguyen-Xuan, 1990] Hoc, J.-M. & Nguyen-Xuan, A. (1990). Chapter 2.3 - language semantics, mental models and analogy. In J.-M. Hoc, T. Green, R. Samurçay, & D. Gilmore (Eds.), *Psychology of Programming* (pp. 139 – 156). London: Academic Press http://www.sciencedirect.com/science/article/pii/B9780123507723500148.

[Hochanadel and Finamore, 2015] Hochanadel, A. & Finamore, D. (2015). Fixed And Growth Mindset In Education And How Grit Helps Students Persist In The Face Of Adversity. *Journal of International Education Research – First Quarter*, *11*(1), 47–51, https://doi.org/10.19030/jier.v11i1.9099.

[Hoffman et al., 2011] Hoffman, D. M., Lu, M., & Pelton, T. (2011). A web-based generation and delivery system for active code reading. In *Proceedings of the 42nd ACM technical symposium on Computer science education - SIGCSE '11* (pp. 483). New York, New York, USA: ACM Press http://portal.acm.org/citation.cfm?doid=1953163.1953301.

[Howe and Berv, 2000] Howe, K. & Berv, J. (2000). Constructing constructivism, epistemological and pedagogical. *Teachers College Record*, *102*(7), 19–40.

[Hsiao et al., 2008] Hsiao, I.-H., Brusilovsky, P., & Sosnovsky, S. (2008). Web-based Parameterized Questions for Object-Oriented Programming. *World Conference on E-Learning, E-Learn 2008*, (pp. 3728–3735).

[Hsiao et al., 2010] Hsiao, I. H., Sosnovsky, S., & Brusilovsky, P. (2010). Guiding students to the right questions: Adaptive navigation support in an E-Learning system for Java programming. *Journal of Computer Assisted Learning*, *26*(4), 270–283, https://doi.org/10.1111/j.1365-2729.2010.00365.x.

[Ingalls, 2020] Ingalls, D. (2020). The evolution of smalltalk: From smalltalk-72 through squeak. *Proc. ACM Program. Lang.*, *4*(HOPL), https://doi.org/10.1145/3386335 https://doi.org/10.1145/3386335.

[Izu et al., 2019] Izu, C., et al. (2019). Fostering program comprehension in novice programmers - learning activities and learning trajectories. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education*, ITiCSE-WGR '19 (pp. 27–52). New York, NY, USA: Association for Computing Machinery https://doi-org.pros.lib.unimi.it/10.1145/3344429.3372501.

[Jacobson, 2000] Jacobson, N. (2000). Using on-computer exams to ensure beginning students' programming competency. *ACM SIGCSE Bulletin*, *32*(4), 53–56, https://doi.org/10.1145/369295.369324.

[Johnson and Soloway, 1984] Johnson, L. W. & Soloway, E. (1984). PROUST: KNOWLEDGE-BASED PROGRAM UNDERSTANDING. In *Proceedings - International Conference on Software Engineering*, number March (pp. 369–380).

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

[Jorion et al., 2015] Jorion, N., Gane, B. D., James, K., Schroeder, L., DiBello, L. V., & Pellegrino, J. W. (2015). An Analytic Framework for Evaluating the Validity of Concept Inventory Claims. *Journal of Engineering Education*, *104*(4), 454–496, https://doi.org/10.1002/jee.20104 https://onlinelibrary.wiley.com/doi/abs/10.1002/jee.20104.

[Kafai et al., 2019] Kafai, Y., Proctor, C., & Lui, D. (2019). From theory bias to theory dialogue: Embracing cognitive, situated, and critical framings of computational thinking in k-12 cs education. In *Proceedings of the 2019 ACM Conference on International Computing Education Research*, ICER '19 (pp. 101–109). New York, NY, USA: Association for Computing Machinery https://doi.org/10.1145/3291279.3339400.

[Kalelioğlu, 2015] Kalelioğlu, F. (2015). A new way of teaching programming skills to K-12 students: Code.org. *Computers in Human Behavior*, *52*(2015), 200–210, https://doi.org/10.1016/j.chb.2015.05.047 http://www.sciencedirect.com/science/article/pii/S0747563215004288.

[Kane, 2016] Kane, M. J. (2016). Current Concerns in Validity Theory. *Language Learning*, *38*(4), 319–342.

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

[Kane and Bejar, 2014] Kane, M. T. & Bejar, I. I. (2014). Cognitive frameworks for assessment, teaching, and learning: A validity perspective. *Psicologia Educativa*, *20*(2), 117–123, https://doi.org/10.1016/j.pse.2014.11.006 http://dx.doi.org/10.1016/j.pse.2014.11.006.

[Karavirta et al., 2015] Karavirta, V., Haavisto, R., Kaila, E., Laakso, M.-J., Rajala, T., & Salakoski, T. (2015). Interactive Learning Content for Introductory Computer Science Course Using the ViLLE Exercise Framework. *2015 International Conference on Learning and Teaching in*

Computing and Engineering, (pp. 9–16)., https://doi.org/10.1109/LaTiCE.2015.24 http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7126224.

[Karpierz and Wolfman, 2014] Karpierz, K. & Wolfman, S. A. (2014). Misconceptions and concept inventory questions for binary search trees and hash tables. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, SIGCSE '14 (pp. 109–114). New York, NY, USA: Association for Computing Machinery https://doi.org/10.1145/2538862.2538902.

[Kay, 2007] Kay, A. (2007). *The Real Computer Revolution Hasn't Happened Yet*. Technical report, VPRI Memo M-2007-007-a, Glendale, CA.

[Kelley, 2007a] Kelley, K. (2007a). Confidence Intervals for Standardized Effect Sizes :. *Journal of Statistical Software*, *20*(8), 1–24, https://doi.org/http://dx.doi.org/10.18637/jss.v020.i08.

[Kelley, 2007b] Kelley, K. (2007b). Methods for the Behavioral, Educational, and Social Sciences: An R package. *Behavior Research Methods*, *39*(4), 979–984, https://doi.org/10.3758/BF03192993 http://www.springerlink.com/index/10.3758/BF03192993.

[Kelly et al., 2018] Kelly, A., Finch, L., Bolles, M., & Shapiro, R. B. (2018). Blockytalky: New programmable tools to enable students' learning networks. *International Journal of Child-Computer Interaction*, *18*, 8–18, https://doi.org/https://doi.org/10.1016/j.ijcci.2018.03.004 https://www.sciencedirect.com/science/article/pii/S2212868918300394.

[Kemeny et al., 1968] Kemeny, J. G., Kurtz, T. E., & Cochran, D. S. (1968). *Basic: a manual for BASIC, the elementary algebraic language designed for use with the Dartmouth Time Sharing System*. Dartmouth Publications.

[Kennedy and Kraemer, 2018] Kennedy, C. & Kraemer, E. T. (2018). What are they thinking?: Eliciting student reasoning about troublesome concepts in introductory computer science. In *Proceedings of the 18th Koli Calling International Conference on Computing Education Research*, Koli Calling '18 (pp. 7:1–7:10). New York, NY, USA: ACM.

[Kimura, 1979] Kimura, T. (1979). Reading before composition. In *Proceedings of the tenth SIGCSE technical symposium on Computer science education - SIGCSE '79* (pp. 162–166). New York, New York, USA: ACM Press http://portal.acm.org/citation.cfm?doid=800126.809575.

[Kingston and Nash, 2011] Kingston, N. & Nash, B. (2011). Formative assessment: A meta-analysis and a call for research. *Educational Measurement: Issues and Practice*, *30*(4), 28–37, https://doi.org/10.1111/j.1745-3992.2011.00220.x.

[Kluger and DeNisi, 1996] Kluger, A. N. & DeNisi, A. (1996). The effects of feedback interventions on performance: A historical review, a meta-analysis, and a preliminary feedback intervention theory. *Psychological Bulletin*, *119*(2), 254–284, https://doi.org/10.1037/0033-2909.119.2.254 http://doi.apa.org/getdoi.cfm?doi=10.1037/0033-2909.119.2.254.

[Knowlton, 1966] Knowlton, K. C. (1966). A programmer's description of l6. *Communications of the ACM*, *9*(8), 616–625.

[Knuth, 1963] Knuth, D. E. (1963). Computer-drawn flowcharts. *Communications of the ACM*, *6*(9), 555–563.

[Kölling et al., 2003] Kölling, M., Quig, B., Patterson, A., & Rosenberg, J. (2003). The BlueJ System and its Pedagogy. *Computer Science Education*, *13*(4), 249–268, https://doi.org/10.1076/csed.13.4.249.17496 http://www.tandfonline.com/doi/abs/10.1076/csed.13.4.249.17496.

[Kölling and Rosenberg, 2001] Kölling, M. & Rosenberg, J. (2001). Guidelines for teaching object orientation with Java. *Proceedings of the Conference on Integrating Technology into Computer Science Education, ITiCSE*, (pp. 33–36)., https://doi.org/10.1145/507758.377461.

[Kollmansberger, 2010] Kollmansberger, S. (2010). Helping students build a mental model of computation. In *Proceedings of the fifteenth annual conference on Innovation and technology in computer science education - ITiCSE '10* (pp. 128). New York, New York, USA: ACM Press http://dl.acm.org/citation.cfm?id=1822127.

[Koltun et al., 1983] Koltun, P., Deimel Jr, L. E., & Perry, J. (1983). Progress report on the study of program reading. *ACM SIGCSE Bulletin*, *15*(1), 168–176.

[Korhonen, 2010] Korhonen, A. (2010). Applications of visual algorithm simulation. In E. M. O. Abu-Taieh & A. A. El-Sheikh (Eds.), *Handbook of Research on Discrete Event Simulation Environments: Technologies and Applications* (pp. 234–251). Hershey, PA, USA: IGI Global http://doi.acm.org/10.1145/2393596.2393624.

[Krapp et al., 1992] Krapp, A., Hidi, S., & Renninger, K. (1992). Factors Affecting Performance in First-year Computing. *The Role of interest in learning and development*, *32*(2), 368.

[Kumar, 2006] Kumar, A. N. (2006). Using Enhanced Concept Map for Student Modeling in Programming Tutors. *Proceedings of the Nineteenth International Florida Artificial Intelligence Research Society Conference*, (pp. 527–532). https://www.aaai.org/Library/FLAIRS/2006/flairs06-103.php.

[Kumar, 2015] Kumar, A. N. (2015). Solving Code-tracing Problems and its Effect on Code-writing Skills Pertaining to Program Semantics. *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education - ITiCSE '15*, (pp. 314–319)., https://doi.org/10.1145/2729094.2742587 http://dl.acm.org/citation.cfm?doid=2729094.2742587.

[Kumar, 2016] Kumar, A. N. (2016). Providing the Option to Skip Feedback in a Worked Example Tutor. In A. Micarelli, J. Stamper, & K. Panourgia (Eds.), *Intelligent Tutoring Systems*, Lecture Notes in Computer Science (pp. 101–110). Cham: Springer International Publishing http://link.springer.com/10.1007/978-3-319-39583-8.

[Kurvinen et al., 2016] Kurvinen, E., Hellgren, N., Kaila, E., Laakso, M.-J., & Salakoski, T. (2016). Programming misconceptions in an introductory level programming course exam. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '16 (pp. 308–313). New York, NY, USA: Association for Computing Machinery https://doi.org/10.1145/2899415.2899447.

[Lammers, 1986] Lammers, S. (1986). Programmers at Work https://programmersatwork.wordpress.com/programmers-at-work-charles-simonyi/.

[Lancaster et al., 2019] Lancaster, T., Robins, A., & Fincher, S. A. (2019). Assessment and plagiarism. In S. A. Fincher & A. V. Robins (Eds.), *The Cambridge handbook of computing education research* (pp. 414–444). Cambridge, UK: Cambridge University Press.

[LaToza et al., 2020] LaToza, T. D., Arab, M., Loksa, D., & Ko, A. J. (2020). Explicit programming strategies. *Empirical Software Engineering*, *25*(4), 2416–2449.

[Lave and Wenger, 1991] Lave, J. & Wenger, E. (1991). *Situated Learning: Legitimate Peripheral Participation*. Cambridge University Press.

[Lecarme, 1974] Lecarme, O. (1974). Structured programming, programming teaching and the language Pascal. *ACM SIGPLAN Notices*, *9*(7), 15–21, https://doi.org/10.1145/953224.953226.

[Lee et al., 2011] Lee, I., et al. (2011). Computational thinking for youth in practice. *ACM Inroads*, *2*(1), 32–37, https://doi.org/10.1145/1929887.1929902.

[Lee and Ko, 2015] Lee, M. J. & Ko, A. J. (2015). Comparing the Effectiveness of Online Learning Approaches on CS1 Learning Outcomes. *Proceedings of the eleventh annual International Conference on International Computing Education Research - ICER '15*, (pp. 237–246)., https://doi.org/10.1145/2787622.2787709 http://dl.acm.org/citation.cfm?id=2787622.2787709.

[Lee et al., 2013] Lee, M. J., Ko, A. J., & Kwan, I. (2013). In-game assessments increase novice programmers' engagement and level completion speed. *Proceedings of the ninth annual international ACM conference on International computing education research - ICER '13*, (pp. 153)., https://doi.org/10.1145/2493394.2493410 http://dl.acm.org/citation.cfm?doid=2493394.2493410.

[Legg and Hookway, 2020] Legg, C. & Hookway, C. (2020). Pragmatism. In E. N. Zalta (Ed.), *The Stanford Encyclopedia of Philosophy*. Standford, CA, USA: Metaphysics Research Lab, Stanford University, fall 2020 edition.

[Leighton and Gierl, 2007] Leighton, J. P. & Gierl, M. J. (2007). Verbal reports as data for cognitive diagnostic assessment. In J. Leighton & M. Gierl (Eds.), *Cognitive Diagnostic Assessment for Education: Theory and Applications* (pp. 146–172). Cambridge: Cambridge University Press https://www.cambridge.org/core/product/identifier/CBO9780511611186A014/type/book{_}part.

[Lemos, 1975] Lemos, R. S. (1975). FORTRAN Programming: An Analysis of Pedagogical Alternatives. *Journal of Educational Data Processing*, *12*(3), 21–29.

[Lemos, 1979] Lemos, R. S. (1979). Teaching programming languages: A survey of approaches. *SIGCSE Bull.*, *11*(1), 174–181, https://doi.org/10.1145/953030.809578 https://doi-org.offcampus.lib.washington.edu/10.1145/953030.809578.

[Lemos, 1980] Lemos, R. S. (1980). Measuring programming language proficiency. *AEDS Journal*, *13*(4), 261–273, https://doi.org/10.1080/00011037.1980.11008280.

[Leppan et al., 2007] Leppan, R., Cilliers, C., & Taljaard, M. (2007). Supporting CS1 with a program beacon recognition tool. *ACM International Conference Proceeding Series*, *226*, 66–75, https://doi.org/10.1145/1292491.1292499.

[Lieberman, 1984] Lieberman, H. (1984). Steps toward better debugging tools for lisp. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, LFP '84 (pp. 247–255). New York, NY, USA: Association for Computing Machinery https://doi.org/10.1145/800055.802041.

[Lishinski et al., 2016] Lishinski, A., Yadav, A., Good, J., & Enbody, R. (2016). Introductory Programming : Gender Differences and Interactive Effects of Students ' Motivation , Goals and Self-Efficacy on Performance. *Proceedings of the 12th International Computing Education Research Conference*, (pp. 211–220)., https://doi.org/10.1145/2960310.2960329.

[Lister, 2016] Lister, R. (2016). Toward a developmental epistemology of computer programming. In *Proceedings of the 11th Workshop in Primary and Secondary Computing Education*, WiPSCE

'16 (pp. 5–16). New York, NY, USA: Association for Computing Machinery https://doi.org/10.1145/2978249.2978251.

[Lister et al., 2004a] Lister, R., et al. (2004a). A multi-national study of reading and tracing skills in novice programmers. In *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education*, ITiCSE-WGR '04 (pp. 119–150). New York, NY, USA: Association for Computing Machinery https://doi.org/10.1145/1044550.1041673.

[Lister et al., 2009] Lister, R., Fidge, C., & Teague, D. (2009). Further evidence of a relationship between explaining, tracing and writing skills in introductory programming. *ACM SIGCSE Bulletin*, *41*(3), 161, https://doi.org/10.1145/1595496.1562930.

[Lister et al., 2004b] Lister, R., et al. (2004b). A multi-national study of reading and tracing skills in novice programmers. *ACM SIGCSE Bulletin*, *36*(4), 119, https://doi.org/10.1145/1041624.1041673 http://portal.acm.org/citation.cfm?doid=1041624.1041673.

[Littlefield et al., 1988] Littlefield, J., Delclos, V., Lever, S., Clayton, K., Bransford, J., & Franks, J. (1988). Learning Logo: Methods of teaching, transfer of general skills, and attitudes toward school and computers. In R. E. Mayer (Ed.), *Learning computer programming: Multiple research perspectives* (pp. 111–135). Hillsdale, NJ: Erlbaum.

[Loboda and Brusilovsky, 2008] Loboda, T. & Brusilovsky, P. (2008). Adaptation in the Context of Explanatory Visualization. *EC-TEL 2008*, *5192*.

[Loboda and Brusilovsky, 2010] Loboda, T. D. & Brusilovsky, P. (2010). User-adaptive explanatory program visualization: Evaluation and insights from eye movements. *User Modeling and User-Adapted Interaction*, *20*(3), 191–226, https://doi.org/10.1007/s11257-010-9077-1.

[Loksa, 2020] Loksa, D. (2020). *Explicitly Training Metacognition and Self-Regulation for Computer Programming*. PhD thesis https://faculty.washington.edu/ajko/dissertations/Loksa2020Dissertation.pdf.

[Loksa and Ko, 2016] Loksa, D. & Ko, A. J. (2016). The role of self-regulation in programming problem solving process and success. In *Proceedings of the 2016 ACM Conference on International Computing Education Research*, ICER '16 (pp. 83–91). New York, NY, USA: ACM http://doi.acm.org/10.1145/2960310.2960334.

[Loksa et al., 2016] Loksa, D., Ko, A. J., Jernigan, W., Oleson, A., Mendez, C. J., & Burnett, M. M. (2016). Programming, problem solving, and self-awareness: Effects of explicit guidance. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, CHI '16 (pp. 1449–1461). New York, NY, USA: ACM http://doi.acm.org/10.1145/2858036.2858252.

[Long et al., 2018] Long, Y., Holstein, K., & Aleven, V. (2018). What exactly do students learn when they practice equation solving? Refining knowledge components with the additive factors model. *ACM International Conference Proceeding Series*, (pp. 399–408)., https://doi.org/10.1145/3170358.3170411.

[Lopez et al., 2008] Lopez, M., Whalley, J., Robbins, P., & Lister, R. (2008). Relationships between reading, tracing and writing skills in introductory programming. In *Proceedings of the Fourth International Workshop on Computing Education Research*, ICER '08 (pp. 101–112). New York, NY, USA: Association for Computing Machinery https://doi.org/10.1145/1404520.1404531.

[Luxton-Reilly, 2016] Luxton-Reilly, A. (2016). Learning to Program is Easy. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education - ITiCSE '16* (pp. 284–289). New York, New York, USA: ACM Press http://dl.acm.org/citation.cfm?doid=2899415.2899432.

[Luxton-Reilly et al., 2018a] Luxton-Reilly, A., et al. (2018a). Developing assessments to determine mastery of programming fundamentals. In *Proceedings of the 2017 ITiCSE Conference on Working Group Reports*, ITiCSE-WGR '17 (pp. 47–69). New York, NY, USA: Association for Computing Machinery https://doi.org/10.1145/3174781.3174784.

[Luxton-Reilly and Petersen, 2017] Luxton-Reilly, A. & Petersen, A. (2017). The compound nature of novice programming assessments. In *Proceedings of the Nineteenth Australasian Computing Education Conference*, ACE '17 (pp. 26–35). New York, NY, USA: Association for Computing Machinery https://doi.org/10.1145/3013499.3013500.

[Luxton-Reilly et al., 2018b] Luxton-Reilly, A., et al. (2018b). Introductory programming: A systematic literature review. In *Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE 2018 Companion (pp. 55–106). New York, NY, USA: Association for Computing Machinery https://doi.org/10.1145/3293881.3295779.

[Malmi et al., 2019] Malmi, L., Sheard, J., Kinnunen, P., Simon, & Sinclair, J. (2019). Computing Education Theories. In *Proceedings of the 2019 ACM Conference on International Computing Education Research*, volume 11 (pp. 187–197). New York, NY, USA: ACM https://dl.acm.org/doi/10.1145/3291279.3339409.

[Maloney et al., 2010] Maloney, J., Resnick, M., & Rusk, N. (2010). The Scratch programming language and environment. *ACM Transactions on Computing Education*, *10*(4), 1–15, https://doi.org/10.1145/1868358.1868363.

[Mann et al., 1994] Mann, L. M., Linn, M. C., & Clancy, M. (1994). Can Tracing Tools Contribute to Programming Proficiency? The LISP Evaluation Modeler. *Interactive Learning Environments*, *4*(1), 96–113, https://doi.org/10.1080/1049482940040104.

[Margulieux, 2019] Margulieux, L. E. (2019). Spatial encoding strategy theory: The relationship between spatial skill and stem achievement. In *Proceedings of the 2019 ACM Conference on International Computing Education Research*, ICER '19 (pp. 81–90). New York, NY, USA: Association for Computing Machinery https://doi.org/10.1145/3291279.3339414.

[Mayer, 1976] Mayer, R. E. (1976). Some conditions of meaningful learning for computer programming: Advance organizers and subject control of frame order. *Journal of Educational Psychology*, *68*(2), 143–150, https://doi.org/10.1037/0022-0663.68.2.143.

[Mayer, 1979a] Mayer, R. E. (1979a). A psychology of learning BASIC. *Communications of the ACM*, *22*(11), 589–593, https://doi.org/10.1145/359168.359171.

[Mayer, 1979b] Mayer, R. E. (1979b). *Analysis of a Simple Computer Programming Language: Transactions, Prestatements and Chunks*. Technical report, Series in Learning and Cognition, Tech. Rep. No. 79-2, U. of California, Santa Barbara, California https://eric.ed.gov/?id=ED207549.

[Mayer, 1979c] Mayer, R. E. (1979c). Twenty years of research on advance organizers: Assimilation theory is still the best predictor of results. *Instructional Science*, *8*(2), 133–167 http://www.jstor.org/stable/23368209.

[Mayer, 1981] Mayer, R. E. (1981). The Psychology of How Novices Learn Computer Programming. *ACM Computing Surveys*, *13*(1), 121–141, https://doi.org/10.1145/356835.356841.

[Mayer, 1982] Mayer, R. E. (1982). *Diagnosis and Remediation of Computer Programming Skill for creative Problem Solving. Volume 1 Description of Research Methods and Results*. Technical report, University of California, Santa Barbara (UCSB).

[Mayer, 1985] Mayer, R. E. (1985). Learning In Complex Domains: A Cognitive Analysis of Computer Programming. *19*(C), 89–130, https://doi.org/10.1016/S0079-7421(08)60525-3.

[Mayrhauser and Vans, 1995] Mayrhauser, A. V. & Vans, a. M. (1995). Program Comprehension During Software Maintenence and Evolution. *Computer*, *28*(8), 44–55, https://doi.org/10.1109/2.402076.

[McCracken et al., 2001] McCracken, M., et al. (2001). A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *ACM SIGCSE Bulletin*, *33*(4), 125, https://doi.org/10.1145/572139.572181 http://0-doi.acm.org.aupac.lib.athabascau.ca/10.1145/572139.572181http://portal.acm.org/citation.cfm?doid=572139.572181.

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

[McKenney and Reeves, 2012] McKenney, S. E. & Reeves, T. C. (2012). *Conducting Educational Design Research*. London: Routledge.

[Mead et al., 2006] Mead, J., Gray, S., Hamer, J., James, R., Sorva, J., Clair, C. S., & Thomas, L. (2006). A cognitive approach to identifying measurable milestones for programming skill acquisition. *ITiCSE-WGR '06*, (December 2006), 182, https://doi.org/10.1145/1189215.1189185 http://portal.acm.org/citation.cfm?doid=1189215.1189185.

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

[Meyers-Levy, 1986] Meyers-Levy, J. (1986). *Gender differences in information processing: A selectivity interpretation*. PhD thesis, Northwestern University Evanston, IL.

[Moreno et al., 2004] Moreno, A., Myller, N., Sutinen, E., & Ben-Ari, M. (2004). Visualizing programs with Jeliot 3. *Proceedings of the working conference on Advanced visual interfaces - AVI '04*, (pp. 373)., https://doi.org/10.1145/989863.989928 http://dl.acm.org/citation.cfm?id=989863.989928.

[Morgado, 2005] Morgado, L. (2005). *Framework for Computer Programming in Preschool and Kindergarten*. PhD thesis https://repositorio.utad.pt/bitstream/10348/5344/1/phd_lcmorgado.pdf.

[Morrison et al., 2015] Morrison, B., Margulieux, L., & Guzdial, M. (2015). Subgoals, Context, and Worked Examples in Learning Computing Problem Solving. In *Proceedings of the eleventh annual International Conference on International Computing Education Research - ICER '15* (pp. 267–268).: ACM Press.

[Moss et al., 2006] Moss, P. A., Girard, B. J., & Haniford, L. C. (2006). Validity in Educational Assessment. *Review of Research in Education*, *30*, 109–162 http://www.jstor.org/stable/4129771.

[Murphy and Thomas, 2008] Murphy, L. & Thomas, L. (2008). Dangers of a fixed mindset: implications of self-theories research for computer science education. *ACM SIGCSE Bulletin*, *40*(3), 271–275, https://doi.org/10.1145/1597849.1384344 http://dl.acm.org/citation.cfm?id=1384271.1384344{%}5Cnpapers3://publication/doi/10.1145/1384271.1384344.

[Myers and Chatlani, 2017] Myers, D. S. & Chatlani, N. (2017). Implementing an adaptive tutorial system for coding literacy. *Consortium for Computing Sciences in Colleges: Southeastern Conference*.

[Myller, 2007] Myller, N. (2007). Automatic Generation of Prediction Questions during Program Visualization. *Electronic Notes in Theoretical Computer Science*, *178*(1), 43–49, https://doi.org/10.1016/j.entcs.2007.01.034 http://dx.doi.org/10.1016/j.entcs.2007.01.034.

[Naps, 2005] Naps, T. (2005). JHAVÉ: Supporting Algorithm Visualization. *IEEE Computer Graphics and Applications*, *25*(5), 49–55, https://doi.org/10.1109/MCG.2005.110 http://ieeexplore.ieee.org/document/1510539/.

[Naps et al., 2002] Naps, T. L., et al. (2002). Exploring the role of visualization and engagement in computer science education. In *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education*, ITiCSE-WGR '02 (pp. 131–152). New York, NY, USA: Association for Computing Machinery https://doi.org/10.1145/960568.782998.

[National Academies of Sciences and Medicine, 2018] National Academies of Sciences, E. & Medicine (2018). *How People Learn II: Learners, Contexts, and Cultures*. Washington, DC: The National Academies Press https://www.nap.edu/catalog/24783/how-people-learn-ii-learners-contexts-and-cultures.

[Nelson et al., 2019] Nelson, G. L., Hu, A., Xie, B., & Ko, A. J. (2019). Towards validity for a formative assessment for language-specific program tracing skills. In *Proceedings of the 19th Koli Calling International Conference on Computing Education Research*, Koli Calling '19 (pp. 20:1–20:10). New York, NY, USA: ACM.

[Nelson and Ko, 2018] Nelson, G. L. & Ko, A. J. (2018). On use of theory in computing education research. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*, ICER '18 (pp. 31–39). New York, NY, USA: Association for Computing Machinery https://doi.org/10.1145/3230977.3230992.

[Nelson et al., 2020] Nelson, G. L., et al. (2020). Differentiated assessments for advanced courses that reveal issues with prerequisite skills: A design investigation. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education*, ITiCSE-WGR

'20 (pp. 75–129). New York, NY, USA: Association for Computing Machinery https://doi.org/10.1145/3437800.3439204.

[Nelson et al., 2017] Nelson, G. L., Xie, B., & Ko, A. J. (2017). Comprehension first: Evaluating a novel pedagogy and tutoring system for program tracing in cs1. In *Proceedings of the 2017 ACM Conference on International Computing Education Research*, ICER '17 (pp. 2–11). New York, NY, USA: Association for Computing Machinery https://doi.org/10.1145/3105726.3106178.

[Nesbit et al., 2015] Nesbit, J., Liu, L., Liu, Q., & Adesope, O. (2015). Work in Progress: Intelligent Tutoring Systems in Computer Science and Software Engineering Education. (pp. 26.1754.1–26.1754.12)., https://doi.org/10.18260/p.25090.

[Newell et al., 1972] Newell, A., Simon, H. A., et al. (1972). *Human problem solving*, volume 104. Prentice-hall Englewood Cliffs, NJ.

[Niss, 2005] Niss, M. (2005). The concept and role of theory in mathematics education. In *Relating pratice and research in Mathematics education: Proceedings of Norma, 2005*, volume 4 (pp. 97–110). http://www.norme.me/NORMA05.pdf.

[O'Brien, 2003] O'Brien, M. (2003). *Software Comprehension - A review and research direction*. Technical Report UL-CSIS-03-3, University of Limerick.

[Palumbo, 1990] Palumbo, D. B. (1990). Programming Language/Problem-Solving Research: A Review of Relevant Issues. *Review of Educational Research*, *60*(1), 65–89, https://doi.org/10.3102/00346543060001065.

[Pane, 2002] Pane, J. F. (2002). *A programming system for children that is designed for usability*. PhD thesis, Carnegie Mellon University.

[Papert, 1971] Papert, S. (1971). *A computer laboratory for elementary schools*. Technical report, Massachusetts Institute of Technology. Artificial Intelligence Laboratory.

[Papert, 1980] Papert, S. (1980). *Mindstorms: Children, Computers, and Powerful Ideas*. New York, NY, USA: Basic Books, Inc.

[Pareja-Flores et al., 2007] Pareja-Flores, C., Urquiza-Fuentes, J., & Velázquez-Iturbide, J. A. (2007). Winhipe: An ide for functional programming based on rewriting and visualization. *ACM SIGPLAN Notices*, *42*(3), 14–23.

[Parker and Guzdial, 2016] Parker, M. C. & Guzdial, M. (2016). Replication, validation, and use of a language independent CS1 knowledge assessment. *ICER*, (pp. 93–101)., https://doi.org/10.1145/2960310.2960316.

[Parker et al., 2016] Parker, M. C., Guzdial, M., & Engleman, S. (2016). Replication, validation, and use of a language independent cs1 knowledge assessment. In *Proceedings of the 2016 ACM Conference on International Computing Education Research*, ICER '16 (pp. 93–101). New York, NY, USA: Association for Computing Machinery https://doi.org/10.1145/2960310.2960316.

[Paul and Vahrenhold, 2013] Paul, W. & Vahrenhold, J. (2013). Hunting high and low: Instruments to detect misconceptions related to algorithms and data structures. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13 (pp. 29–34). New York, NY, USA: Association for Computing Machinery https://doi.org/10.1145/2445196.2445212.

[Pausch et al., 2000] Pausch, R., Dann, W., & Cooper, S. (2000). Alice : a 3-D Tool for Introductory Programming Concepts. *Journal of Computing Sciences in Colleges*, *15*(5), 107–116.

[Pea, 1986] Pea, R. D. (1986). Language-Independent conceptual "bugs" in novice programming. *Journal of Educational Computing Research*, *2*(1), 25–36.

[Pears et al., 2007] Pears, A., et al. (2007). A survey of literature on the teaching of introductory programming. In *Working Group Reports on ITiCSE on Innovation and Technology in Computer Science Education*, ITiCSE-WGR '07 (pp. 204–223). New York, NY, USA: ACM http://doi.acm.org/10.1145/1345443.1345441.

[Peffer et al., 2016] Peffer, M., Renken, M., & Tomanek, D. (2016). Practical strategies for collaboration across discipline-based education research and the learning sciences. *CBE—Life Sciences Education*, *15*(4), es11, https://doi.org/10.1187/cbe.15-12-0252.

[Pelchen et al., 2020] Pelchen, T., Mathieson, L., & Lister, R. (2020). On the evidence for a learning hierarchy in data structures exams. In *Proceedings of the Twenty-Second Australasian Computing Education Conference*, ACE'20 (pp. 122–131). New York, NY, USA: Association for Computing Machinery https://doi.org/10.1145/3373165.3373179.

[Perkins and Martin, 1985] Perkins, D. & Martin, F. (1985). *Fragile Knowledge and Neglected Strategies in Novice Programmers. IR85-22.* Technical report https://www.semanticscholar.org/paper/Fragile-Knowledge-and-Neglected-Strategies-in-Perkins-Martin/18659022c6ccafa941d857936f13891d24bbec46.

[Petersen et al., 2011] Petersen, A., Craig, M., & Zingaro, D. (2011). Reviewing cs1 exam question content. In *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education*, SIGCSE '11 (pp. 631–636). New York, NY, USA: Association for Computing Machinery https://doi.org/10.1145/1953163.1953340.

[Phillips, 1995] Phillips, D. C. (1995). The good, the bad, and the ugly: The many faces of constructivism. *Educational researcher*, *24*(7), 5–12.

[Phillips, 2000] Phillips, D. C. (2000). *Constructivism in Education: Opinions and Second Opinions on Controversial Issues. Ninety-Ninth Yearbook of the National Society for the Study of Education.* ERIC.

[Piech and Gregg, 2018] Piech, C. & Gregg, C. (2018). BlueBook: A Computerized Replacement for Paper Tests in Computer Science. In *SIGCSE '18* (pp. 562–567).: ACM Press https://doi.org/10.1145/3159450.3159587.

[Pollock et al., 2019] Pollock, J., Roesch, J., Woos, D., & Tatlock, Z. (2019). Theia: automatically generating correct program state visualizations. In *Proceedings of the 2019 ACM SIGPLAN Symposium on SPLASH-E - SPLASH-E 2019*, volume 19 (pp. 46–56). New York, New York, USA: ACM Press https://doi.org/10.1145/3358711.3361625.

[Porter et al., 2019] Porter, L., Zingaro, D., Liao, S. N., Taylor, C., Webb, K. C., Lee, C., & Clancy, M. (2019). Bdsi: A validated concept inventory for basic data structures. In *Proceedings of the 2019 ACM Conference on International Computing Education Research*, ICER '19 (pp. 111–119). New York, NY, USA: Association for Computing Machinery https://doi.org/10.1145/3291279.3339404.

[PracticeIt, 2016] PracticeIt (2016). http://practiceit.cs.washington.edu. Accessed: 2016-12-12.

[Prior, 2003] Prior, J. C. (2003). Online Assessment of SQL Query Formulation Skills. *Ace '03*, *20*, 247–256 http://delivery.acm.org/10.1145/860000/858433/p247-prior.pdf.

[Qian and Lehman, 2017] Qian, Y. & Lehman, J. (2017). Students' misconceptions and other difficulties in introductory programming: A literature review. *ACM Trans. Comput. Educ.*, *18*(1), https://doi.org/10.1145/3077618 https://doi.org/10.1145/3077618.

[Ralston, 1971] Ralston, A. (1971). Fortran and the First Course in Computer Science. *Acm Sigcse*, *3*(4), 24–29, https://doi.org/10.1145/382214.382499.

[Ramalingam et al., 2004] Ramalingam, V., LaBelle, D., & Wiedenbeck, S. (2004). Self-Efficacy and Mental Models in Learning to Program. *Proceedings of the 9th annual SIGCSE conference on Innovation and technology in computer science education - ITiCSE '04*, *36*(3), 171–175, https://doi.org/10.1145/1007996.1008042 http://portal.acm.org/citation.cfm?doid=1007996.1008042.

[Ramalingam and Wiedenbeck, 1999] Ramalingam, V. & Wiedenbeck, S. (1999). Development and validation of scores on a computer programming self-efficacy scale and group analyses of

novice programmer self-efficacy. *Journal of Educational Computing Research*, *19*(4), 367–381, https://doi.org/10.2190/C670-Y3C8-LTJ1-CT3P.

[Reeves, 2006] Reeves, T. C. (2006). Design research from the technology perspective. In *Educational design research* (pp. 86–109). Routledge.

[Reges, 2006] Reges, S. (2006). Back to basics in CS1 and CS2. *ACM SIGCSE Bulletin*, *38*(1), 293, https://doi.org/10.1145/1124706.1121432.

[Rich et al., 2017] Rich, K. M., Strickland, C., Binkowski, T. A., Moran, C., & Franklin, D. (2017). K-8 learning trajectories derived from research literature: Sequence, repetition, conditionals. In *ICER '17* (pp. 182–190).: ACM.

[Riese, 2017] Riese, E. (2017). Students' Experience and Use of Assessment in an Online Introductory Programming Course. In *2017 International Conference on Learning and Teaching in Computing and Engineering (LaTICE)* (pp. 30–34).: IEEE http://ieeexplore.ieee.org/document/8064428/.

[Rifkin and Deimel, 1994] Rifkin, S. & Deimel, L. (1994). Applying Program Comprehension Techniques to Improve Software Inspections. In *19th Annual NASA Software Engineering Laboratory Workshop*.

[Rifkin and Deimel, 2000] Rifkin, S. & Deimel, L. (2000). Program comprehension techniques improve software inspections: A case study. In *Proceedings - IEEE Workshop on Program Comprehension*, volume 2000-Janua (pp. 131–138).: IEEE Computer Society.

[Roberts, 2004] Roberts, E. (2004). The dream of a common language. *Proceedings of the 35th SIGCSE technical symposium on Computer science education - SIGCSE '04*, *36*(1), 115, https://doi.org/10.1145/971300.971343 http://dl.acm.org/citation.cfm?id=971300.971343.

[Robins et al., 2003] Robins, A., Rountree, J., & Rountree, N. (2003). Learning and Teaching Programming: A Review and Discussion. *Computer Science Education*, *13*(2), 137–172, https://doi.org/10.1076/csed.13.2.137.14200 http://www.tandfonline.com/doi/abs/10.1076/csed.13.2.137.14200.

[Rowe and Smaill, 2007] Rowe, G. & Smaill, C. (2007). Development of an electromagnetic course—concept inventory—a work in progress. In *Proceedings of the eighteenth Conference of Australian Association for Engineering* The University of Melbourne, Melbourne, Australia: Department of Computer Science and Software Engineering.

[Salac et al., 2020] Salac, J., Thomas, C., Butler, C., Sanchez, A., & Franklin, D. (2020). Tippamp;see: A learning strategy to guide students through use - modify scratch activities. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, SIGCSE '20 (pp. 79–85). New York, NY, USA: Association for Computing Machinery https://doi.org/10.1145/3328778.3366821.

[Sanders et al., 2013] Sanders, K., et al. (2013). The canterbury questionbank: Building a repository of multiple-choice cs1 and cs2 questions. In *Proceedings of the ITiCSE Working Group Reports Conference on Innovation and Technology in Computer Science Education-Working Group Reports*, ITiCSE -WGR '13 (pp. 33–52). New York, NY, USA: Association for Computing Machinery https://doi.org/10.1145/2543882.2543885.

[Sands et al., 2018] Sands, D., Parker, M., Hedgeland, H., Jordan, S., & Galloway, R. (2018). Using concept inventories to measure understanding. *Higher Education Pedagogies*, *3*(1), 173–182, https://doi.org/10.1080/23752696.2018.1433546 https://www.tandfonline.com/doi/abs/10.1080/23752696.2018.1433546.

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

[Santos and Correia, 2020] Santos, J. & Correia, F. F. (2020). A Review of Pattern Languages for Software Documentation. *ACM International Conference Proceeding Series*, (pp. 6–10)., https://doi.org/10.1145/3424771.3424786.

[Schulte et al., 2010] Schulte, C., Clear, T., Taherkhani, A., Busjahn, T., & Paterson, J. H. (2010). An introduction to program comprehension for computer science educators. *ITiCSE-WGR '10*, (pp.65)., https://doi.org/10.1145/1971681.1971687 http://portal.acm.org/citation.cfm?doid=1971681.1971687.

[Schumacher and Czerwinski, 1992] Schumacher, R. & Czerwinski, M. (1992). Mental models and the acquisition of expert knowledge.

[Schurdak, 1965] Schurdak, J. J. (1965). *AN APPROACH TO THE USE OF COMPUTERS IN THE INSTRUCTIONAL PROCESS AND EVALUATION*. Technical Report 1, IBM.

[Schurdak, 1967] Schurdak, J. J. (1967). An Approach to the Use of Computers in the Instructional Process and an Evaluation. *American Educational Research Journal*, *4*(1), 59–73, https://doi.org/10.3102/00028312004001059 http://aer.sagepub.com/cgi/doi/10.3102/00028312004001059.

[Scott and Ghinea, 2014] Scott, M. J. & Ghinea, G. (2014). On the Domain-Specificity of mindsets: The relationship between aptitude beliefs and programming practice. *IEEE Trans. Educ.*, *57*(3), 169–174.

[Scriven, 1967] Scriven, M. (1967). The methodology of evaluation. In R. Tyler, R. M. Gagné, & M. Scriven (Eds.), *Perspectives of curriculum evaluation* (pp. 39–85). Chicago: Rand McNally Education.

[Sessoms and Henson, 2018] Sessoms, J. & Henson, R. A. (2018). Applications of diagnostic classification models: A literature review and critical commentary. *Measurement*, *16*(1), 1–17.

[Shaffer et al., 2011] Shaffer, C., Karavirta, V., Korhonen, A., & Naps, T. (2011). Opendsa: Beginning a community active-ebook project. In *Proceedings of the 11th Koli Calling International Conference on Computing Education Research, Joensuu, Finland, November 17-20, 2011* (pp. 112–117). Joensuu, Finland: University of Eastern Finland.

[Sheard et al., 2016] Sheard, J., Souza, D. D., Klemperer, P., Porter, L., & Zingaro, D. (2016). Benchmarking Introductory Programming Exams: Some Preliminary Results. *ICER '16*, (pp. 103–111)., https://doi.org/10.1145/2899415.2899473.

[Shih and Alessi, 1993] Shih, Y. F. & Alessi, S. M. (1993). Mental models and transfer of learning in computer programming. *Journal of Research on Computing in Education*, *26*(2), 154–175, https://doi.org/10.1080/08886504.1993.10782084.

[Shinners-Kennedy, 2008] Shinners-Kennedy, D. (2008). *The Everydayness of Threshold Concepts*, (pp. 119–128). BRILL: Leiden, The Netherlands https://brill.com/view/book/edcoll/9789460911477/BP000010.xml.

[Shneiderman, 1977] Shneiderman, B. (1977). Teaching programming: A spiral approach to syntax and semantics. *Computers & Education*, *1*(4), 193–197, https://doi.org/10.1016/0360-1315(77)90008-2 http://linkinghub.elsevier.com/retrieve/pii/0360131577900082.

[Shneiderman and Mayer, 1979] Shneiderman, B. & Mayer, R. (1979). Syntactic/semantic interactions in programmer behavior: A model and experimental results. *International Journal of Computer & Information Sciences*, *8*(3), 219–238, https://doi.org/10.1007/BF00977789 http://link.springer.com/10.1007/BF00977789.

[Shute et al., 2008] Shute, V. J., Hansen, E. G., & Almond, R. G. (2008). You can't fatten a hog by weighing it – or can you? Evaluating an assessment for learning system called ACED. *International Journal of Artificial Intelligence in Education*, *18*(4), 289–316 http://myweb.fsu.edu/vshute/pdf/shute2008{_}a.pdf.

[Simon et al., 2016] Simon, et al. (2016). Benchmarking Introductory Programming Exams. In *ITiCSE '16*, volume 159 (pp. 154–159).: ACM Press http://doi.acm.org/10.1145/2899415.2899473http://dl.acm.org/citation.cfm?doid=2899415.2899473.

[Simon et al., 2010] Simon, B., Clancy, M., McCartney, R., Morrison, B., Richards, B., & Sanders, K. (2010). Making sense of data structures exams. In *Proceedings of the Sixth International Workshop on Computing Education Research*, ICER '10 (pp. 97–106). New York, NY, USA: Association for Computing Machinery https://doi.org/10.1145/1839594.1839612.

[Sirkiä, 2018] Sirkiä, T. (2018). Jsvee & kelmu: Creating and tailoring program animations for computing education. *Journal of Software: Evolution and Process*, *30*(2), e1924.

[Sirkiä and Haaranen, 2017] Sirkiä, T. & Haaranen, L. (2017). Improving online learning activity interoperability with acos server. *Software: Practice and Experience*, *47*(11), 1657–1676, https://doi.org/10.1002/spe.2492 https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2492.

[Sjoerdsma, 1976] Sjoerdsma, T. (1976). An interactive pseudo-assembler for introductory computer science. *ACM SIGCSE Bulletin*, *8*(1), 342–349, https://doi.org/10.1145/952989.803496 http://portal.acm.org/citation.cfm?doid=952989.803496.

[Sleeman et al., 1986] Sleeman, D., Putnam, R. T., Baxter, J., & Kuspa, L. (1986). Pascal and high school students: A study of errors. *Journal of Educational Computing Research*, *2*(1), 5–23, https://doi.org/10.2190/2XPP-LTYH-98NQ-BU77.

[Snow et al., 2017] Snow, E., Rutstein, D., Bienkowski, M., & Xu, Y. (2017). Principled assessment of student learning in high school computer science. In *Proceedings of the 2017 ACM Conference on International Computing Education Research*, ICER '17 (pp. 209–216). New York, NY, USA: Association for Computing Machinery https://doi.org/10.1145/3105726.3106186.

[Solar-lezama, 2008] Solar-lezama, A. (2008). *Program Synthesis by Sketching*. PhD thesis, University of California-Berkeley.

[Solomon et al., 2020] Solomon, C., et al. (2020). History of logo. *Proc. ACM Program. Lang.*, *4*(HOPL), https://doi.org/10.1145/3386329 https://doi.org/10.1145/3386329.

[Solomon, 1976] Solomon, C. J. (1976). Leading a child to a computer culture. In *Proceedings of the ACM SIGCSE-SIGCUE Technical Symposium on Computer Science and Education*, SIGCSE '76 (pp. 79–83). New York, NY, USA: Association for Computing Machinery https://doi.org/10.1145/800107.803452.

[Soloway and Ehrlich, 1984] Soloway, E. & Ehrlich, K. (1984). Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, *SE-10*(5), 595–609, https://doi.org/10.1109/TSE.1984.5010283.

[Sorva, 2010] Sorva, J. (2010). Reflections on threshold concepts in computer programming and beyond. *Koli Calling '10*, (pp. 21–30)., https://doi.org/10.1145/1930464.1930467 http://dl.acm.org/citation.cfm?id=1930464.1930467.

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

[Sorva, 2013a] Sorva, J. (2013a). Notional machines and introductory programming education. *ACM Trans. Comput. Educ*, *13*(8), 31, https://doi.org/10.1145/2483710.2483713 http://dx.doi.org/10.1145/2483710.2483713.

[Sorva, 2013b] Sorva, J. (2013b). *Visual Program Simulation in Introductory Program Education*, volume 53.

[Sorva et al., 2013a] Sorva, J., Karavirta, V., & Malmi, L. (2013a). A Review of Generic Program Visualization Systems for Introductory Programming Education. *ACM Transactions on Computing Education*, *13*(4), 15.1 – 15.64, https://doi.org/10.1145/2490822.

[Sorva et al., 2013b] Sorva, J., Karavirta, V., & Malmi, L. (2013b). A review of generic program visualization systems for introductory programming education. *Trans. Comput. Educ.*, *13*(4), 15:1–15:64, https://doi.org/10.1145/2490822.

[Sorva et al., 2013c] Sorva, J., Lönnberg, J., & Malmi, L. (2013c). Students' ways of experiencing visual program simulation. *Computer Science Education*, *23*(3), 207–238, https://doi.org/10.1080/08993408.2013.807962 https://doi.org/10.1080/08993408.2013.807962.

[Sorva and Seppälä, 2014] Sorva, J. & Seppälä, O. (2014). Research-based design of the first weeks of CS1. *Proceedings of the 14th Koli Calling International Conference on Computing Education Research (Koli Calling '14)*, (November 2014), 71–80, https://doi.org/10.1145/2674683.2674690 http://dl.acm.org/citation.cfm?id=2674690.

[Sorva and Sirkia, 2010] Sorva, J. & Sirkia, T. (2010). UUhistle: a software tool for visual program simulation. *Proceedings of the 10th Koli Calling International Conference on Computing Education Research Koli Calling 10*, (pp. 49–54)., https://doi.org/10.1145/1930464.1930471 http://dl.acm.org/citation.cfm?id=1930464.1930471.

[Sorva and Sirkiä, 2011] Sorva, J. & Sirkiä, T. (2011). Context-sensitive guidance in the UUhistle program visualization system. *Proceedings of the Sixth Program Visualization Workshop (PVW 2011)*, (pp. 77–85).

[Sorva and Sirkiä, 2015] Sorva, J. & Sirkiä, T. (2015). Embedded questions in ebooks on programming — Useful for a) summative assessment , b) formative assessment , or c) something else? *Proceedings of the 15th Koli Calling Conference on Computing Education Research*, (pp. 152–156)., https://doi.org/10.1145/2828959.2828961.

[Spohrer et al., 1985] Spohrer, J. C., Soloway, E., & Pope, E. (1985). A goal/plan analysis of buggy pascal programs. *Hum.-Comput. Interact.*, *1*(2), 163–207, https://doi.org/10.1207/s15327051hci0102_4.

[Stahl et al., 2006] Stahl, G., Koschmann, T., & Suthers, D. (2006). Cambridge Handbook of the learning sciences. Computer-supported collaborative learning: An historical perspective. *Cambridge handbook of the learning sciences*, (pp. 409–426)., https://doi.org/10.1145/1124772.1124855 http://gerrystahl.net/hci/chls.pdf.

[Stegeman, 2019] Stegeman, M. (2019). A set of exercises and tests for teaching tracing skills using a mastery approach. In *ACM International Conference Proceeding Series*, volume 13 (pp. 1–40).: Research and Practice in Technology Enhanced Learning https://doi.org/10.1145/3364510.3366154.

[Strauss and Smith, 2009] Strauss, M. E. & Smith, G. T. (2009). Construct validity: Advances in theory and methodology. *Annual Review of Clinical Psychology*, *5*, 1–25, https://doi.org/10.1146/annurev.clinpsy.032408.153639 /pmc/articles/PMC2739261/?report=abstracthttps://www.ncbi.nlm.nih.gov/pmc/articles/PMC2739261/.

[Strömbäck et al., 2019] Strömbäck, F., Mannila, L., Asplund, M., & Kamkar, M. (2019). A student's view of concurrency - a study of common mistakes in introductory courses on concurrency. In *Proceedings of the 2019 ACM Conference on International Computing Education Research*, ICER '19 (pp. 229–237). New York, NY, USA: Association for Computing Machinery https://doi.org/10.1145/3291279.3339415.

[Sudol-Delyser et al., 2012] Sudol-Delyser, L. A., Stehlik, M., & Carver, S. (2012). Code comprehension problems as learning events. *Annual Conference on Innovation and Technology in Computer Science Education, ITiCSE*, (pp. 81–86)., https://doi.org/10.1145/2325296.2325319.

[Suppes et al., 1977] Suppes, P., Smith, R., & Beard, M. (1977). University-level computer-assisted instruction at Stanford: 1975. *Instructional Science*, *6*(2), 151–185, https://doi.org/10.1007/BF00121084 http://link.springer.com/10.1007/BF00121084.

[Sweller, 1994] Sweller, J. (1994). Cognitive load theory, learning difficulty, and instructional design. *Learning and Instruction*, *4*(4), 295–312, https://doi.org/10.1016/0959-4752(94)90003-5.

[Sweller, 2010] Sweller, J. (2010). Element interactivity and intrinsic, extraneous, and germane cognitive load. *Educational Psychology Review*, *22*(2), 123–138, https://doi.org/10.1007/s10648-010-9128-5.

[Syang and Dale, 1993] Syang, A. & Dale, N. B. (1993). Computerized adaptive testing in computer science: Assessing student programming abilities. In *Proceedings of the Twenty-Fourth SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '93 (pp. 53–56). New York, NY, USA: Association for Computing Machinery https://doi.org/10.1145/169070.169109.

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

[Taylor and Dionne, 2000] Taylor, K. L. & Dionne, J.-P. (2000). Accessing problem-solving strategy knowledge: The complementary use of concurrent verbal protocols and retrospective debriefing. *J. Educ. Psychol.*, *92*(3), 413.

[Teague and Lister, 2015] Teague, D. & Lister, R. (2015). *Neo-Piagetian Theory and the Novice Programmer (Note: This is actually a PhD thesis written by Donna Teague, under the supervision of Raymond Lister)*. PhD thesis.

[Teague et al., 2015] Teague, D., Lister, R., & Ahadi, A. (2015). Mired in the web: Vignettes from charlotte and other novice programmers. In D. D'Souza & K. Falkner (Eds.), *17th Australasian Computing Education Conference (ACE 2015)*, volume 160 of *CRPIT* (pp. 165–174). Sydney, Australia: ACS http://crpit.com/confpapers/CRPITV160Teague.pdf.

[Teasley, 1993] Teasley, B. M. (1993). Program comprehension skills and their acquisition: A call for an ecological paradigm. In E. Lemut, B. du Boulay, & G. Dettori (Eds.), *Cognitive Models and Intelligent Environments for Learning Programming* (pp. 71–79). Berlin, Heidelberg: Springer Berlin Heidelberg.

[Tew and Dorn, 2013] Tew, A. E. & Dorn, B. (2013). The Case for Validated Tools in Computer Science Education Research. *Computer*, *46*(9), 60–66, https://doi.org/10.1109/MC.2013.259 http://ieeexplore.ieee.org/document/6562691/.

[Tew and Guzdial, 2010] Tew, A. E. & Guzdial, M. (2010). Developing a validated assessment of fundamental cs1 concepts. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, SIGCSE '10 (pp. 97–101). New York, NY, USA: Association for Computing Machinery https://doi.org/10.1145/1734263.1734297.

[Tew and Guzdial, 2011] Tew, A. E. & Guzdial, M. (2011). The fcs1: A language independent assessment of cs1 knowledge. In *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education*, SIGCSE '11 (pp. 111–116). New York, NY, USA: Association for Computing Machinery https://doi.org/10.1145/1953163.1953200.

[Thomas et al., 2019] Thomas, A., Frank-Bolton, P., Stopera, T., & Simha, R. (2019). Stochastic tree-based generation of program-tracing practice questions. *SIGCSE 2019 - Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, (pp. 91–97)., https://doi.org/10.1145/3287324.3287492.

[Torlak and Bodik, 2014] Torlak, E. & Bodik, R. (2014). A lightweight symbolic virtual machine for solver-aided host languages. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI '14* New York, New York, USA: ACM Press http://dl.acm.org/citation.cfm?doid=2594291.2594340.

[Tsai et al., 2019] Tsai, M. J., Wang, C. Y., & Hsu, P. F. (2019). Developing the Computer Programming Self-Efficacy Scale for Computer Literacy Education. *Journal of Educational Computing Research*, *56*(8), 1345–1360, https://doi.org/10.1177/0735633117746747 https://doi.org/10.1177/0735633117746747.

[Turbak et al., 1999] Turbak, F., Royden, C., Stephan, J., & Herbst, J. (1999). Teaching recursion before loops in CS1. *Journal of Computing in Small Colleges*, *14*(May), 86–101 http://cs.wellesley.edu/{~}fturbak/pubs/jcsc99.pdf.

[Vagianou, 2006] Vagianou, E. (2006). Program working storage: A beginner's model. In *Proceedings of the 6th Baltic Sea Conference on Computing Education Research: Koli Calling 2006*, Baltic Sea '06 (pp. 69–76). New York, NY, USA: Association for Computing Machinery https://doi.org/10.1145/1315803.1315816.

[Valstar et al., 2019] Valstar, S., Griswold, W. G., & Porter, L. (2019). The relationship between prerequisite proficiency and student performance in an upper-division computing course. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, SIGCSE '19 (pp. 794–800). New York, NY, USA: Association for Computing Machinery https://doi.org/10.1145/3287324.3287419.

[Van Merrienboer and Krammer, 1987] Van Merrienboer, J. J. G. & Krammer, H. P. M. (1987). Instructional strategies and tactics for the design of introductory computer programming courses in

high school. *Instructional Science*, *16*(3), 251–285, https://doi.org/10.1007/BF00120253 http://link.springer.com/10.1007/BF00120253.

[Venables et al., 2009] Venables, A., Tan, G., & Lister, R. (2009). A Closer Look at Tracing, Explaining and Code Writing Skills in the Novice Programmer. *Proceedings of the Fifth International Workshop on Computing Education Research Workshop - ICER '09*, (2009), 117–128, https://doi.org/10.1145/1584322.1584336 http://portal.acm.org/citation.cfm?doid=1584322.1584336.

[Viera and Garrett, 2005] Viera, A. J. & Garrett, J. M. (2005). Understanding interobserver agreement: the kappa statistic. *Family medicine*, *37 5*, 360–3.

[Virtanen et al., 2005] Virtanen, A., Lahtinen, E., & Jarvinen, H.-M. (2005). VIP, a Visual Interpreter for Learning Introductory Programming with C++. *Koli Calling 2005 Conference on Computer Science Education*, (November), 125–130 http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.101.9111{&}rep=rep1{&}type=pdf{#}page=120.

[Von Glasersfeld, 2012] Von Glasersfeld, E. (2012). *A constructivist approach to teaching*. Routledge.

[Warth et al., 2008] Warth, A., Yamamiya, T., Ohshima, Y., & Wallace, S. (2008). Toward a more scalable end-user scripting language. In *Sixth International Conference on Creating, Connecting and Collaborating through Computing (C5 2008)* (pp. 172–178).: IEEE.

[Watson et al., 2014] Watson, C., Li, F. W. B., & Godwin, J. L. (2014). No tests required: comparing traditional and dynamic predictors of programming success. *Proceedings of the 45th ACM technical symposium on Computer science education - SIGCSE '14*, (pp. 469–474)., https://doi.org/10.1145/2538862.2538930 http://dl.acm.org/citation.cfm?id=2538930.

[Weber and Brusilovsky, 2001] Weber, G. & Brusilovsky, P. (2001). ELM-ART: An Adaptive Versatile System for Web-based Instruction. *International Journal of Artificial Intelligence in Education (IJAIED)*, *12*(June 2014), 351–384 http://art.ph-freiburg.de/Lisp-Course.

[Weber and Brusilovsky, 2016] Weber, G. & Brusilovsky, P. (2016). ELM-ART – An Interactive and Intelligent Web-Based Electronic Textbook. *International Journal of Artificial Intelligence in Education*, *26*(1), 72–81, https://doi.org/10.1007/s40593-015-0066-8 http://link.springer.com/10.1007/s40593-015-0066-8.

[Whalley et al., 2006] Whalley, J. L., Lister, R., Thompson, E., Clear, T., Robbins, P., Kumar, P. K. A., & Prasad, C. (2006). An australasian study of reading and comprehension skills in novice programmers, using the bloom and solo taxonomies. In *Proceedings of the 8th Australasian*

Conference on Computing Education - Volume 52, ACE '06 (pp. 243–252). AUS: Australian Computer Society, Inc.

[Wiedenbeck, 1986] Wiedenbeck, S. (1986). Beacons in computer program comprehension. *International Journal of Man-Machine Studies*, *25*(6), 697–709, https://doi.org/https://doi.org/10.1016/S0020-7373(86)80083-9 https://www.sciencedirect.com/science/article/pii/S0020737386800839.

[Wilson and Shrock, 2001] Wilson, B. C. & Shrock, S. (2001). Contributing to success in an introductory computer science course: a study of twelve factors. *ACM SIGCSE Bulletin*, *33*(1), 184–188, https://doi.org/http://doi.acm.org/10.1145/366413.364581 http://portal.acm.org/citation.cfm?doid=364447.364581{%}5Cnhttp://ezproxy.stevens.edu:2077/citation.cfm?id=364447.364581{&}coll=ACM{&}dl=ACM{&}CFID=90624379{&}CFTOKEN=25082170.

[Winslow, 1996] Winslow, L. E. (1996). Programming Pedagogy - A Psychological Overview. *ACM SIGCSE Bulletin*, *28*(3), 17–22, https://doi.org/10.1145/234867.234872 http://portal.acm.org/citation.cfm?doid=234867.234872.

[Wood et al., 1976] Wood, D., Bruner, J., & Ross, G. (1976). The role of tutoring in problem solving. *Journal of child psychology and psychiatry, and allied disciplines*, *17 2*, 89–100.

[Woodman et al., 1999] Woodman, M., Griffiths, R., Macgregor, M., & Holland, S. (1999). OU LearningWorks: A customized programming environment for Smalltalk modules. *Proceedings - International Conference on Software Engineering*, (January 1999), 638–641, https://doi.org/10.1109/icse.1999.841064.

[Woods and Warren, 1995] Woods, P. J. & Warren, J. R. (1995). Rapid Prototyping of an Intelligent Tutorial System. *Ascilite '95*, (pp. 557–563). http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.89.4475&rep=rep1&type=pdf.

[Xie et al., 2019a] Xie, B., Davidson, M. J., Li, M., & Ko, A. J. (2019a). An item response theory evaluation of a Language-Independent CS1 knowledge assessment. In *SIGCSE '19*: ACM.

[Xie et al., 2019b] Xie, B., et al. (2019b). A theory of instruction for introductory programming skills. *Computer Science Education*, *29*(2-3), 205–253, https://doi.org/10.1080/08993408.2019.1565235.

[Xie et al., 2018] Xie, B., Nelson, G. L., & Ko, A. J. (2018). An explicit strategy to scaffold novice program tracing. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, SIGCSE '18 (pp. 344–349). New York, NY, USA: Association for Computing Machinery https://doi.org/10.1145/3159450.3159527.

[Yamamiya, 2003] Yamamiya, T. (2003). LanguageGame - an interactive parser generator. In *Proceedings of the Conference on Creating, Connecting and Collaborating Through Computing (C5) 2003* (pp. 110–110).: IEEE.

[Yehezkel, 2003] Yehezkel, C. (2003). Making program execution comprehensible one level above the machine language. *ITiCSE '03 Proceedings of the 8th annual conference on Innovation and technology in computer science education*, (pp. 124)., https://doi.org/10.1145/961290.961547 http://dl.acm.org/citation.cfm?id=961290.961547.

[Zander et al., 2008] Zander, C., Boustedt, J., Eckerdal, A., Mccartney, R., Moström, J., Ratcliffe, M., & Sanders, K. (2008). *Threshold Concepts In Computer Science: A Multi-National Empirical Investigation*, (pp. 105–118). BRILL https://brill.com/view/book/edcoll/9789460911477/BP000009.xml.

[Zavala and Mendoza, 2018] Zavala, L. & Mendoza, B. (2018). On the Use of Semantic-Based AIG to Automatically Generate Programming Exercises. *Proceedings of the 49th ACM Technical Symposium on Computer Science Education - SIGCSE '18*, (pp. 14–19)., https://doi.org/10.1145/3159450.3159608 http://dl.acm.org/citation.cfm?doid=3159450.3159608.

[Zhan et al., 2019] Zhan, P., Jiao, H., Man, K., & Wang, L. (2019). Using JAGS for Bayesian Cognitive Diagnosis Modeling: A Tutorial. *Journal of Educational and Behavioral Statistics*, (pp. 107699861982604)., https://doi.org/10.3102/1076998619826040 http://journals.sagepub.com/doi/10.3102/1076998619826040.

[Zhi et al., 2015] Zhi, J., Garousi-Yusifolu, V., Sun, B., Garousi, G., Shahnewaz, S., & Ruhe, G. (2015). Cost, benefits and quality of software development documentation: A systematic mapping. *Journal of Systems and Software*, *99*(2015), 175–198, https://doi.org/10.1016/j.jss.2014.09.042.

[Zingaro et al., 2012] Zingaro, D., Petersen, A., & Craig, M. (2012). Stepping up to integrative questions on cs1 exams. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*, SIGCSE '12 (pp. 253–258). New York, NY, USA: Association for Computing Machinery https://doi.org/10.1145/2157136.2157215.

[Zur-Bargury et al., 2013] Zur-Bargury, I., Pârv, B., & Lanzberg, D. (2013). A nationwide exam as a tool for improving a new curriculum. In *ITiCSE '18* (pp. 267–272).: ACM.

Appendix A

APPENDIX FOR DIFFERENTIATED ASSESSMENTS SKILLS ANALYSIS AND ASSESSMENT QUESTIONS

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

A.1 Study of Prerequisite Skills

In order to better connect our qualitative coding categories to existing knowledge, and as a way of validating our categories, we associated each of our codes (Section 5.3.1, Tables 5.1 to 5.6) with topics in the ACM Computing Curriculum Guidelines [Association for Computing Machinery, 2013], prior work by Goldman et al. [Goldman et al., 2010] to identify prerequisite topics, and Misconception Catalogue compiled by Sorva [Sorva, 2012].

A.1.1 ACM CC2013

The ACM 2013 Curriculum Guide [Joint Task Force on Computing Curricula, Association for Computing Machinery (ACM) and IEEE Computer Society, 2013] consists of 18 Knowledge Areas in computing.

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

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

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

The Body of Knowledge is a specification of the content to be covered and a curriculum is an implementation for it. However, Computer Science, unlike many technical disciplines, does not have a well-described list of topics that appear in virtually all introductory courses [Association for Computing Machinery, 2013]. Many of them focus on topics such as Software Development Fundamentals, Programming Languages, and Software Engineering. Some courses start with object-oriented programming, while others use functional programming. In addition, it is not said that all Software Development Fundamentals should be covered in a first course. In practice, however, most fundamental topics are typically covered in CS1.

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

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

- Differences among best, expected, and worst case behaviors of an algorithm
- Asymptotic analysis of upper and expected complexity bounds
- Big O notation: formal definition
- Complexity classes, such as constant, logarithmic, linear, quadratic, and exponential
- Empirical measurements of performance
- Time and space trade-offs in algorithms

As we can see, many of these topics are taught in a course called Data Structures and Algorithms, or similar, which also has a prerequisite course (e.g., CS1 or CS2). Thus, we are not going to list all

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

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

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

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

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

- Construct and debug programs using the standard libraries available with a chosen programming language.

A.1.2 Core Concepts Identified by Experts

Goldman et al. [Goldman et al., 2008, Goldman et al., 2010] set out to create a concept inventory for introductory computing subjects. An important part of this process is to investigate which core concepts are typically covered in introductory courses, and which of those are perceived to be important and difficult. This allows the final concept inventory to focus on the most important concepts that students are most likely to find difficult, thus keeping the size of the concept inventory down.

We present a summary of the final concepts for programming fundamentals below as that is the subject most relevant to this report. The remaining concepts can be found in the original paper [Goldman et al., 2010]. To find the most difficult and important concepts, the authors represented each concept as a point on a 2D plane, the x coordinate being the mean importance, and the y coordinate being the mean difficulty. The concepts closest to the maximum point $(10, 10)$ were then deemed to be the most important and difficult topics. The top 11 such topics are marked with an asterisk below.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

A.1.3 Misconception Catalogue

Misconceptions can be caused by a lack of knowledge of the syntax, not knowing how a particular syntactical construct behaves (i.e., due to an incorrect or incomplete notional machine [du Boulay, 1986, Sorva, 2012]), or not knowing how to use a particular construct in order to solve a programming problem (i.e. lacking strategic knowledge). Several misconceptions in introductory programming have been identified and addressed in the literature [Pea, 1986, Sorva, 2012, Kurvinen et al., 2016, Qian and Lehman, 2017]. For example, a classic syntactical misconception is the use of an assignment operator (=) instead of the comparison operator (==). At a conceptual level (notional machine), an example of a misconception is that a variable can hold more than one value; this is manifested in the task of swapping two variables. With respect to the strategic level, novices find modularization and decomposition, general abstraction, testing, and debugging very difficult [Goldman et al., 2008].

In this report we do not contribute new misconceptions but instead use existing knowledge about misconceptions to check that our new assessment questions might feasibly catch frequent misconceptions. We wanted to check how misconceptions from the research literature aligned with our prerequisite skills coding. However, as the number of studies related to misconceptions is too large to be cited her, we chose to use the Misconception Catalogue collected by Sorva. It represents a review of literature from the past forty years [Sorva, 2012]. Although, it gives examples of novice programmers' misconceptions about the content of introductory programming courses in general, it is somewhat leaning towards misconceptions found in courses taking Object-Oriented approach.

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

1. General (the overall nature of programs and program execution),
2. VarAssign (variables, assignment and expression evaluation),
3. Control (flow of control, selection and interation),
4. Calls (subprogram invocations and parameter passing),
5. Rec (recursion),
6. Refs (references and pointers, reference assignment and object identity),
7. ObjClass (the object–class relationship and instantiation),
8. ObjState (object state and attributes),
9. Methods (issues specific to methods and methods calls),
10. OtherOOP (other topics specific to object-oriented programming), and

11. Misc (none of the above).

For examples of particular misconceptions mapped to our codebook, see Tables 5.1 to 5.6.

M Modified Questions

M.1 Data Structure Question (Queue)

M.1.1 Original question

Consider the following data structure:

```
1 public class Y<Key extends Comparable<Key>>
2 {
3     private Key[] A = (Key[]) new Comparable[1];
4     private int lo, hi, N;
5     public void insert(Key in)
6     {
7         A[hi] = in;
8         hi = hi + 1;
9         if (hi == A.length) hi = 0;
10        N = N + 1;
11        if (N == A.length) rebuild();
12    }
13    public Key remove() // assumes Y is not empty
14    {
15        Key out = A[lo];
16        A[lo] = null;
17        lo = lo + 1;
18        if (lo == A.length) lo = 0;
19        N = N - 1;
20        return out;
21    }
22    private void rebuild()
23    {
24        Key[] tmp =
25            (Key[]) new Comparable[2*A.length];
26        for (int i = 0; i < N; i++ )
27            tmp[i] = A[(i + lo) % A.length];
28        A = tmp;
29        lo = 0;
30        hi = N;
31    }
32 }
```

(a) Class Y behaves like which well-known data structure?

A Stack

B Queue

C Priority queue

D Union-find

(b) Write the body of a method `int size()` that returns the number of elements in the data structure.

A `return N;`

B `return A.length;`

C `return A[N];`

D `return hi - lo;`

(c) Which invariant does the data structure maintain after every public operation?

A `N < A.length`

B `lo < hi`

C `hi < N`

D `hi == N`

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

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

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

A $\sim 4N$

B 2

C $\sim 2N$

D 7

(f) How many array accesses does a single call to the most expensive public method of `Y` take in the worst case?

A linear in k.

B constant.

C linearithmic in k.

D quadratic in k.

(g) What is the number of array accesses per operation in the following sequence of $2k$ operations, starting from an empty data structure: `y.insert(1); y.remove(); y.insert(2); y.remove(); y.insert(3); y.remove();` ... `y.insert(k); y.remove();`

A linear in k in the worst case and in the amortized case.

B constant in the worst case.

C constant in the amortized case, but linear in k in the worst case.

D quadratic in k in the worst case.

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

M.1.2 Analysis of the original question

This question assesses the following skills categorized in this paper:

- Simple statements
- Operators
- Assignments
- Tracing
- Conditionals
- Loop constructs
- Array iteration
- Types
- Values and references
- Arrays

- Parameters
- Return values
- Function scoping and data flow
- Classes/records/ADT
- Reasoning about constraints
- Meta-tracing knowledge

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

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

More in-depth analysis on the original exercise:

- The answer to the first answer could be wrong for very different reasons:
  - you do not know these data structures, you are not able to differentiate them (advanced concept)
  - you do not understand the code
- If you know the difference between the mentioned data structures, a very general understanding of the code would be enough to answer item (a): you can exclude union-find because is a totally different setting; you can exclude priority queues because there are no comparisons; lines 8 and 17 should be enough to understand that insertion and removal occur in different places so the stack can also be excluded.
- If one does not understand the circular nature of this queue implementation, they might wrongly select option D for item (b) and option B for item (c). The relevant lines of code here are 9, 18, 27. However, the circularity is not explicitly assessed, since one could correctly answer to items (b) and (c) by considering only lines 10 and 19, without understanding the circularity.
- Item (c) addresses the possible confusion between the length of the array and the number of elements currently in the queue.
- To answer item (d) correctly you need to understand both the circularity and the rebuilding policy.
- To understand the rebuilding policy you need conditionals and array knowledge (line 11).
- Neither the circularity nature of the queue nor implementation and the rebuilding policy are assessed explicitly (separately).

- In item (d), extreme situations may occur: on the one hand, one can answer correctly by tracing the code line by line, without understanding what is going on on an abstract level, on the other hand, if one has already understood how the queue is implemented and they are able to reason about it at a high level, they could answer item (b) without considering the code at all.
- Items from (e) openly address advanced concepts (related to complexity) that still require referring to the code and considering its execution.

M.1.3 Summary of assessed skills

This exercise assesses the following advanced learning outcomes:

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

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

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

M.1.4 New version

(Changes are highlighted in bold)

(a) Class Y behaves like which well-known data structure?

A Stack

B Queue

C Priority queue

D Union find

(b) Write the body of a method `int size()` that returns the number of elements in the data structure.

A `return N;`

B `return A.length;`

C `return A[N];`

D `return hi - lo;`

(c) Which invariant does the data structure maintain after every public operation?

A `N < A.length`

B `lo < hi`

C `hi < N`

D `hi == N`

(c-1) Assume that:
`A` holds $\{3,8,4,1\}$**,**
`lo` holds 3,
`hi` holds 2 and
`N` holds 2.

A Is the above situation something that can occur by calling a sequence of `insert` and `remove`? If yes, give such a sequence, otherwise explain why not.

B What are the contents of `A`, `lo` and `hi` after executing `rebuild`?

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

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

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

```
1 Y y = new Y();
2 Y z = new Y();
3 Y w = z;
4 w.insert(3);
5 z.insert(1);
6 y.insert(2);
7 int a = z.remove();
8 int b = y.remove();
```

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

A $\sim 4N$

B 2

C $\sim 2N$

D 7

(f) How many array accesses does a single call to the most expensive public method of `Y` take in the worst case?

A linear in k.

B constant.

C linearithmic in k.

D quadratic in k.

(g) What is the number of array accesses per operation in the following sequence of $2k$ operations, starting from an empty data structure: `y.insert(1); y.remove(); y.insert(2); y.remove(); y.insert(3); y.remove();` ... `y.insert(k); y.remove();`

A linear in k in the worst case and in the amortized case.

B constant in the worst case.

C constant in the amortized case, but linear in k in the worst case.

D quadratic in k in the worst case.

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

M.1.5 Modifications in the new version

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

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

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

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

- The use of "comparable" (lines 24-25) could prevent comprehension of the declaration. This issue was not classified by our code-book since it is language-specific. However, we added a comment before line 24 to make this clear:

  The line below is essentially:
  `Key[] tmp = new Key[2*A.length];`
  with keys being comparable.

  The fact that key are comparable could be also removed, but that would weaken distractor C of item (a).

- A new question is added as (c-1), in order to assess prerequisites on operators (modulus), conditionals, arrays (indexing and storage), and array iteration.

  The correct answer is: `A` holds $\{1,3,0,0,0,0,0,0\}$, `lo` holds 0, `hi` holds 2, `N` holds 2.

  The second part requires tracing. The instance is not consistent, and this might raise doubts in most conscious students, so we added the idem before, which also assesses their ability to reason about constraints (pre- and post-conditions). A good answer for this item is to notice

that `N` is always equal to the distance (in absolute value) between `hi` and `lo`, which is not true in this instance. Answering correctly to original items (b) and (c) is a good step towards this understanding.

Thus the question addresses circularity, since in this instance `lo` is higher that `hi`.

- We modified question (d) to asses separately the "conditionals" prerequisite skill. Namely we added this question: *How many times was "rebuild" called?*
- Finally, we added another item, (d-1), after (d) to assessing "references" and "instances". The correct answer is `a` holds 1 and `b` holds 2. If they say `a` is 1, they do not know reference semantics; if the say `b` is 3, they do not differentiate instances (they have only *one* queue).

M.2 Concurrency 1

M.2.1 Original question

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

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

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

1. Is *busy-wait* used somewhere in the implementation? If so, where?

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

3. After using the data structure for a while, some users notice that the same idea has been used multiple times (i.e. multiple calls to `idea_get` returned the same idea). Furthermore, ideas sometimes disappear from the buffer, even though `idea_add` indicates success by returning `true`.

   Explain with an example what could have happened when...

   (a) ...the same idea was used multiple times.

   (b) ...the buffer ”lost” one or more ideas.

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

5. Use suitable synchronization primitives to synchronize the code based on the critical sections you found.

   Note: Strive for a solution that allows maximum theoretical parallellism, even though that solution may perform worse in practice due to synchronization overheads (please note if you think this is the case).

   Note: Points may be deducted for excessive locking.

```
#define BUFFER_SIZE 32

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

```
44   // and look through the array until we find a
45   // non-empty element.
46   int pos = rand() % BUFFER_SIZE;
47   while (buffer->ideas[pos] == NULL) {
48     pos = (pos + 1) % BUFFER_SIZE;
49   }
50   // Remove it.
51   const char *result = buffer->ideas[pos];
52   buffer->ideas[pos] = NULL;
53   return result;
54 }
```

M.2.2 Analysis of the original question

This question assesses the following skills categorized in this paper:

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

M.2.3 New version

(Changes highlighted in bold)

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

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

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

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

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

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

```
1 struct idea_buffer a, b;
2 idea_init(&a);
3 idea_init(&b);
4 idea_add(&a, "a");
5 idea_get(&b); // <-- here?
```

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

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

5. After using the data structure for a while, some users notice that the same idea has been used multiple times (i.e. multiple calls to `idea_get` returned the same idea). Furthermore, ideas sometimes disappear from the buffer, even though `idea_add` indicates success by returning `true`.

   Explain with an example what could have happened when...

(a) ...the same idea was used multiple times.

(b) ...the buffer "lost" one or more ideas.

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

7. **Which variables are not shared between threads?**

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

9. Use suitable synchronization primitives to synchronize the code based on the critical sections you found.

   Note: Strive for a solution that allows maximum theoretical parallellism, even though that solution may perform worse in practice due to synchronization overheads (please note if you think this is the case).

   Note: Points may be deducted for excessive locking.

```
1  #define BUFFER_SIZE 32
2
3  struct idea_buffer {
4    // All ideas in the buffer. Empty elements are
5    // set to NULL.
6    const char *ideas[BUFFER_SIZE];
7    // Number of ideas in the buffer.
8    int count;
9  };
10 // Initialize the buffer.
11 void idea_init(struct idea_buffer *buffer) {
12   for (int i = 0; i < BUFFER_SIZE; i++)
13     buffer->ideas[i] = NULL;
14   buffer->count = 0;
15 }
16 // Add a new idea to an empty location in the
17 // buffer. Returns 'false' if the buffer is full.
18 bool idea_add(struct idea_buffer *to,
19               const char *idea) {
20   // Find an empty location.
21   int found = BUFFER_SIZE;
22   for (int i = 0; i < BUFFER_SIZE; i++) {
23     if (to->ideas[i] == NULL) {
```

```
24        found = i;
25        break;
26      }
27    }
28    // Full?
29    if (found >= BUFFER_SIZE)
30      return false;
31    // Insert into the buffer.
32    to->ideas[found] = idea;
33    to->count++;
34    return true;
35  }
36  // Get and remove a random element from the
37  // buffer. If no elements are present, the
38  // function waits for an element to be added.
39  const char *idea_get(struct idea_buffer *from) {
40    while (from->count == 0)
41      ;
42    from->count--;
43    // Find an element. Start from a random index,
44    // and look through the array until we find a
45    // non-empty element.
46    int pos = rand() % BUFFER_SIZE;
47    while (from->ideas[pos] == NULL) {
48      pos = (pos + 1) % BUFFER_SIZE;
49    }
50    // Remove it.
51    const char *result = from->ideas[pos];
52    from->ideas[pos] = NULL;
53    return result;
54  }
```

M.2.4 Modifications in the new version

In the new version, we made the following changes:

- The names of the pointer variables were altered to make it impossible to rely entirely on pattern matching in order to arrive at conclusions regarding shared and non-shared variables in parts 1, 2 and 5.
- Part 1 was added, which explicitly assesses that students understand references in C.
- Part 2 was added, which explicitly assesses the *object* category, that the student understands the difference between struct declarations and instances.

- Part 6 was added, which explicitly assesses whether students understand indirection. Since the pointer variables are renamed, students need to be aware that the different variables actually refer to the same instance.
- Part 7 was added, which assesses *function scoping and data flow* by asking the student to note which variables are not shared, which requires the student to understand which variables are local to functions and which are not.

M.3 Concurrency 2

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

M.3.1 Original question

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

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

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

  You may assume that `wait` is only called once for each time `spawn` is called.

Correct any synchronization issues in the implementation.

```
1  // Function doing the heavy computations. We
2  // want to run this in parallel in two threads.
3  int do_work(int param) {
4      // Here we're doing heavy work...
5
6      // Hint, try uncommenting the following
7      // line to see the problems occurring
8      // more frequently.
9      // timer_msleep(param);
10
11     // For simplicity we simply square
12     // the parameter.
13     return param * param;
14 }
15
16 // Data structure keeping track of a thread
17 // running "do_work".
18 struct work_data {
19     // Parameter to be passed to "do_work".
```

```
20      int param;
21
22      // Result from "do_work" in case
23      // the thread is done.
24      int result;
25  };
26
27  // The first function executed in new threads.
28  void thread_main(struct work_data *data) {
29      data->result = do_work(data->param);
30  }
31
32  // Start a new thread running the function
33  // "do_work" with "param" as a parameter.
34  // Returns a "struct work_data" that may be
35  // passed to "wait" in order to get the result.
36  struct work_data *spawn(int param) {
37      // Allocate a new data structure and
38      // initialize it.
39      struct work_data *data =
40          malloc(sizeof(struct work_data));
41      data->param = param;
42
43      // Create a new thread running "thread_main"
44      // and give it access to "data".
45      thread_new(&thread_main, data);
46
47      return data;
48  }
49
50  // Wait for a thread started with "spawn" to
51  // complete, and get the result produced. "wait"
52  // will also free "data", so we assume that "wait"
53  // is only called once for each call to "spawn".
54  int wait(struct work_data *data) {
55      // Get the result, free the memory
56      // and return it.
57      int result = data->result;
58      free(data);
59      return result;
60  }
61
62  // Main function. If the implementation above is
63  // correct you should not need to change anything
```

```
64 // here. It could be interesting to modify "main"
65 // in order to test your implementation.
66 int main(void) {
67     struct work_data *a = spawn(10);
68     struct work_data *b = spawn(100);
69
70     int c = do_work(5);
71
72     printf("Result for 'a': %d\n", wait(a));
73     printf("Result for 'b': %d\n", wait(b));
74     printf("Result for 'c': %d\n", c);
75
76     return 0;
77 }
```

M.3.2 Analysis of the original question

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

M.3.3 Summary of assessed skills

This question assesses the following skills categorized in this paper:

- Simple statements
- Assignments
- Types
- Values and references
- Indirection
- Parameters
- Return values
- Function scoping and data flow
- Classes/records/ADT
- Object/instance/variable

- Meta-tracing knowledge

Advanced skills:

- Threads
- Semaphores or condition variables

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

M.3.4 New version

(Changes are highlighted in bold)

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

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

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

  You may assume that `wait` is only called once for each time `spawn` is called.

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

```
1 // Function doing the heavy computations. We
2 // want to run this in parallel in two threads.
3 int do_work(int param) {
4     // Here we're doing heavy work...
5
6     // Hint, try uncommenting the following
7     // line to see the problems occurring
8     // more frequently.
9     // timer_msleep(param);
10
11    // For simplicity we simply square
12    // the parameter.
13    return param * param;
14 }
15
16 // Data structure keeping track of a thread
17 // running "do_work".
18 struct work_data {
19    // Parameter to be passed to "do_work".
20    int param;
21
22    // Result from "do_work" in case
23    // the thread is done.
24    int result;
25 };
26
27 // The first function executed in new threads.
28 void thread_main(struct work_data *data) {
29    printf("New thread computing %d\n",
30        data->result);
31    data->result = do_work(data->param);
32 }
33
34 // Initialize data.
35 void initialize_data(int param,
36        struct work_data *init) {
37    init->param = param;
38    init->result = 0;
39 }
40
41 // Start a new thread running the function
42 // "do_work" with "param" as a parameter.
43 // Returns a "struct work_data" that may be
```

```
44 // passed to "wait" in order to get the result.
45 struct work_data *spawn(int param) {
46     // Allocate a new data structure and
47     // initialize it.
48     struct work_data *data =
49         malloc(sizeof(struct work_data));
50     initialize_data(param, data);
51     printf("Initialized data for %d\n",
52         data->param);
53
54     // Create a new thread running "thread_main"
55     // and give it access to "data".
56     thread_new(&thread_main, data);
57
58     return data;
59 }
60
61 // Version of spawn.
62 struct work_data *spawn2(int param,
63         struct work_data *data) {
64     initialize_data(param, data);
65     printf("Initialized data for %d\n",
66         data->param);
67
68     // Create a new thread running "thread_main"
69     // and give it access to "data".
70     thread_new(&thread_main, data);
71
72     return data;
73 }
74
75 // Wait for a thread started with "spawn" to
76 // complete, and get the result produced. "wait"
77 // will also free "data", so we assume that "wait"
78 // is only called once for each call to "spawn".
79 int wait(struct work_data *wait_for) {
80     // Get the result, free the memory
81     // and return it.
82     int result = wait_for->result;
83     free(wait_for);
84     return result;
85 }
86
87 // Main function. If the implementation above is
```

```
88 // correct you should not need to change anything
89 // here. It could be interesting to modify "main"
90 // in order to test your implementation.
91 int main(void) {
92     struct work_data *a = spawn(10);
93     struct work_data *b = spawn(100);
94     // b = spawn_2(1000, b);
95
96     int c = do_work(5);
97
98     printf("Result for 'a': %d\n", wait(a));
99     printf("Result for 'b': %d\n", wait(b));
100    printf("Result for 'c': %d\n", c);
101
102    return 0;
103 }
```

M.3.5 Modifications in the new version

In the new version we made the following changes:

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

M.4 Advanced OOP: Inheritance and Polymorphism

This question contains a piece of code that defines a number of classes in Eiffel, and asks the student what would happen when a piece of provided code is executed. Interestingly, Eiffel allows for *co-variant* overloading of methods, and still it considers the derived type as conforming to the base one (this is in contrast with the type systems of many popular languages, such as Java and C++). The assessment is designed to make students consider the problems this possibility might cause (since Liskov's conditions do not necessarily hold) in a system in which both the base and the derived component are used; see [Meyer, 1995] for the background and the inspiration of the exercise and further discussion.

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

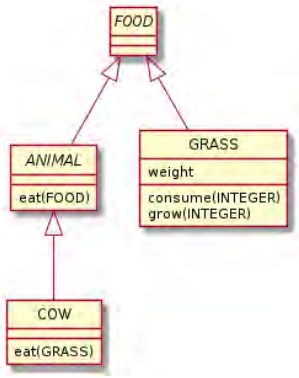


Figure A.1: UML Class diagram for assessment classes

M.4.1 Original question

Consider the Eiffel code below and answer the following questions. An UML class diagram is depicted in Figure A.1.

1. consider the assignment `f := g` at line 80 of feature `make` in the class `APPLICATION`. What happens if after that statement one puts a call `c.eat(f)`? Does it cause an error? If yes, explain whether it is a compile-time or a run-time error.

2. Consider an assignment `f := c`. What happens if after that statement one puts a call `a.eat (f)`? Does it cause an error? If yes, explain whether it is a compile-time or a run-time error.

3. Consider an assignment `f := a`. What happens if after that statement one puts a call `a.eat (f)`? Does it cause an error? If yes, explain whether it is a compile-time or a run-time error.

4. Suggest sensible invariants for class `GRASS` and pre-/post-conditions for features `grow` and `consume`.

5. Suggest a sensible pre-condition for feature `eat` in class `COW` and explain why that would not be effective.

6. Rewrite `eat` in `COW` such that it raises an exception in case (5). Is it a good idea according to a Design By Contract approach?

```
1  deferred class
2    FOOD
3  end
4
5  deferred class ANIMAL inherit FOOD
6  feature
7
8    eat (f: FOOD)
9      require
10       -- f should not refer to me...
11       no_autophagy: f /= ({FOOD} [Current])
12     deferred
13     end
14
15 end
16
17 class GRASS inherit FOOD
18     redefine
19       out
20     end
21
22 feature
23
24   out: STRING
25     do
26       Result := "a bunch of grass (" + weight.out + "kg)"
27     end
28
29   consume (q: INTEGER)
30     do
31       weight := weight - q
32     end
```

```
33
34    grow (q: INTEGER)
35      do
36        weight := weight + q
37      end
38
39    weight: INTEGER
40
41  end
42
43  class COW inherit ANIMAL
44      redefine
45        eat,
46        out
47      end
48
49  feature
50
51    eat (g: GRASS)
52      do
53        g.consume (10)
54      end
55
56    out: STRING
57      do
58        Result := "a cow"
59      end
60
61  end
62
63  class APPLICATION
64  create
65    make
66
67  feature -- Main
68
69    make
70        -- Run application.
71      local
72        a: ANIMAL
73        c: COW
74        g: GRASS
75        f: FOOD
76      do
77        create c
78        create g
79        a := c
80        f := g -- focus on this
81        print (a.out + " is going to eat: " + f.out + "%N")
82        a.eat (f)
```

```
83      end
84
85  end
```

M.4.2 Answers to the original question

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

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

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

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

```
1  class GRASS inherit FOOD -- only the relevant code is reported here
2
3  feature
4    consume (q: INTEGER)
5      require
6        q > 0
7        weight >= q
8      do
9        weight := weight - q
10     ensure
11       weight = old weight - q
12     end
13
14   grow (q: INTEGER)
15     require
16       q > 0
17     do
18       weight := weight + q
19     ensure
20       weight = old weight + q
21     end
22
23 invariant
24   weight >= 0
```

```
25
26 end
```

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

```
1  class COW inherit ANIMAL -- only the relevant code is reported here
2      redefine
3        eat
4      end
5
6  feature
7
8    eat (g: GRASS)
9      require else
10       g.weight >= 10
11     do
12       g.consume (10)
13     end
14
15 end
```

6. One could add a check:

```
1  class COW inherit ANIMAL -- only the relevant code is reported here
2      redefine
3        eat
4      end
5
6  feature
7
8    eat (g: GRASS)
9      do
10       check g >= 10
11       g.consume (10)
12     end
13
14 end
```

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

```
1  class COW inherit ANIMAL -- only the relevant code is reported here
2      redefine
3        eat
4      end
5
6  feature
7
8    eat (g: GRASS)
9      do
10         if g.weight < 10 then
11           g.grow (10 - g.weight)
12         end
13         g.consume (10)
14     end
15
16 end
```

M.4.3 Summary of assessed skills

Although the goal of the exercise is to test the understanding of the co-variant overloading and its relationship with static typing constraints and Liskov's substitution principle (in a correct system, a component C' can be substituted to component C only if the pre-conditions for the use of C' are weaker or equal than the pre-conditions of C, and the post-conditions of C' are stronger or equal than the post-conditions of C), the answering student must master at least the following fundamental skills (from our code-book in Section 5.3).

- Simple Statements
- Operators
- Assignments
- Tracing
- Types
- Values and references
- Parameters
- Classes/records/ADT
- Meta-tracing knowledge

If some of these prerequisites are not clear, wrong answers cannot be clearly attributed to misconceptions in the advanced topics. In particular, a familiarity with a tracing as a general strategy to understand how the interpreter executes the code is needed, but this strategy has to transferred

on a new level, since it should applied not only on (abstract) states, but also to types and method dispatching.

M.4.4 New version

To address some of these issues, the following changes could be made:

1. Add a new concrete `FOOD` class (e.g., `PLANKTON`) and create an object `p` from this class;

2. add a method `log_food` with a parameter `x` of type `FOOD`, the method just prints the dynamic type of the actual parameter bound to `x`;

3. add a question about the output of `log_food(p)`, `log_food(g)`, `log_food(f)`, where `g` is a reference to a `GRASS` object and `f` is a reference to a `GRASS` object of `FOOD` static type (using `f` as an actual parameter of `log_food` before assigning it to a concrete object is not legal in Eiffel, unless the type is marked explicitly as `detachable`, see `log_food2`);

```
1  -- ANIMAL, COW, GRASS, and FOOD as before
2
3  class PLANKTON inherit FOOD -- new class
4    redefine
5      out
6    end
7
8    feature
9
10     out: STRING
11     do
12       Result := "Lots of plankster"
13     end
14 end
15
16 class APPLICATION -- modified
17
18   feature -- Main
19
20     make
21         -- Run application.
22       local
23         a: ANIMAL
24         c: COW
25         g: GRASS
26         f: FOOD
27         p: PLANKTON
28       do
29         create c
30         create g
```

```
31          create p
32          g.grow (5)
33          a := c
34
35              -- log_food2(f) -- log_food(f) not legal
36          f := g
37
38          log_food(p)
39          log_food(g)
40          log_food(f)
41
42          print (a.out + " is going to eat: " + f.out + "%N")
43          a.eat (f)
44          print ("Finished!%N")
45        end
46
47      log_food(x: FOOD)
48      do
49        print ("The food x is: " + x.out + "%N")
50      end
51
52      log_food2(x: detachable FOOD)
53      do
54        if attached x then
55            print ("The food x is: " + x.out + "%N")
56        else
57          print("No x%N")
58
59        end
60      end
61
62  end
```

M.5 BDSI: B.6

We modified some questions from the Basic Data Structures Inventory (BDSI) [Porter et al., 2019], for the purposes of exploring potential changes to question designs, which might also explicitly assess prerequisite skills. However, we have not done any empirical validity experiments on these modified versions yet, to see how actual learners respond to them - for example, having learners think aloud as they answer the question. Thus, we do not really know how good they are yet. If you want to maintain the validity argument for the BDSI (especially if you are using the BDSI for summative use, like to assess learning at the end of a class), do not substitute or add these modified questions, and do not use them as practice or for class assignments, as they are too close to the BDSI questions.

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

M.5.1 Original question

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

```
1 DEFINE mystery(value)
2  current = head
3  temp = nil
4  WHILE current != nil AND current.item != value DO
5   temp = current
6   current = current.next
7  ENDWHILE
8  RETURN temp
9 ENDDEF
```

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

Select one of the following statements:

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

B It returns the node containing `value`, or it returns the last node of the list if `value` is not found.

C It returns the node containing `value`, or it returns `nil` if `value` is not found.

D It returns the node before the one containing `value`, or it returns `nil` if `value` is in the head, or it returns `nil` if `value` is not found.

M.5.2 Analysis of the original question

This question assesses the following prerequisite skills:

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

M.5.3 New version

(Changes are highlighted in bold)

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

```
1 DEFINE mystery(value)
2  current = head
3  temp = nil
4  WHILE current != nil AND current.item != value DO
5   temp = current
6   current = current.next
7  ENDWHILE
8  RETURN temp
9 ENDDEF
```

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

Select all that apply:

A **`mystery` returns the node containing `value`, or it returns the last node of the list of `value` is not in the list.**

B **For the `SinglyLinkedList` holding 2, then 4, then 6, `mystery(1)` will return the last node in the list.**

C **If the value is not in the list, `mystery` returns `nil`.**

D **If the value is in the list, `mystery` returns the node containing the value.**

E **If the value is in the list, `mystery` returns the value contained in the node in the list.**

F **If the value is in the head node of the list, `mystery` returns nil.**

M.5.4 Modifications in the new version

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

M.6 BDSI: B.8

We modified a question from the Basic Data Structures Inventory (BDSI) [Porter et al., 2019], for the purposes of exploring potential changes to question designs, which would add additional required knowledge to the question. However, we have not done any empirical validity experiments on these modified versions yet, to see how actual learners respond to them - for example, having learners think aloud as they answer the question. Thus, we do not really know how good they are yet. If you want to maintain the validity argument for the BDSI (especially if you are using the BDSI for summative use, like to assess learning at the end of a class), do not substitute or add these modified questions, and do not use them as practice or for class assignments, as they are too close to the BDSI questions.

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

M.6.1 Original question

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

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

Select one:

A `ArrayList` is best as it guarantees constant-time access.

B `ArrayList` is best as it guarantees access time proportional to $\log N$ using binary search.

C Unsorted `DoublyLinkedList` is best as it guarantees constant-time access.

D Unsorted `DoublyLinkedList` is best as it guarantees access time proportional to $\log N$ using binary search.

E Sorted `DoublyLinkedList` is best as it guarantees constant-time access.

F Sorted `DoublyLinkedList` is best as it guarantees access time proportional to $\log N$ using binary search.

M.6.2 Analysis of the original question

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

M.6.3 New version

(Changes are highlighted in bold)

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

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

A `ArrayList` is best as it guarantees constant-time access.

B `ArrayList` is best as it guarantees access time proportional to $\log N$ using binary search.

C Unsorted `DoublyLinkedList` is best as it guarantees constant-time access.

D Unsorted `DoublyLinkedList` is best as it guarantees access time proportional to $\log N$ using binary search.

E Sorted `DoublyLinkedList` is best as it guarantees constant-time access.

F Sorted `DoublyLinkedList` is best as it guarantees access time proportional to $\log N$ using binary search.

M.6.4 Modifications in the new version

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

O Other Questions

O.1 Scientific Computing

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

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

O.2 Software Design Question 1

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

Recall the `Pharmacy` and `PharmacyDB` classes from the project:

```
1  public class Pharmacy {
2      private String id;
3      private String owner;
4      private String busName;
5      private String address;
6      private String suite;
7      private String city;
8      private String state;
9      private String zip;
10     private String phone;
11     private String type;
12
13     public Pharmacy() {}
14
15     public Pharmacy(String id, String owner, String busName, String
           address, String suite, String city, String state, String zip,
           String phone, String type) {
16         this.id = id;
17         this.owner = owner;
18         this.busName = busName;
19         this.address = address;
20         this.suite = suite;
21         this.city = city;
22         this.state = state;
23         this.zip = zip;
```

```
24          this.phone = phone;
25          this.type = type;
26      }
27
28      public String getId() {
29          return id;
30      }
31
32      public void setId(String id) {
33          this.id = id;
34      }
35
36      public String getOwner() {
37          return owner;
38      }
39
40      public void setOwner(String owner) {
41          this.owner = owner;
42      }
43
44  // getters and setters for many instance variables are not shown to
      save space
45      public String getZip() {
46          return zip;
47      }
48
49      public void setZip(String zip) {
50          this.zip = zip;
51      }
52
53      public String getPhone() {
54          return phone;
55      }
56
57      public void setPhone(String phone) {
58          this.phone = phone;
59      }
60  }
```

This is a shortened version of PharmacyDB:

```
1  public class PharmacyDB {
2      private HashMap<String, Pharmacy> pharmMap = new HashMap<String,
          Pharmacy>();
3      public void add(Pharmacy pharm)  {
```

```
4           pharmMap.putIfAbsent(pharm.getId(), pharm);
5       }
6       public Pharmacy getPharmById(String id) {
7          return pharmMap.get(id);
8       }
9       public Boolean containsId(String id) {
10         return pharmMap.containsKey(id);
11      }
12
13      /**
14       * return a list of pharmacies sorted by zip code
15       */
16      public List<Pharmacy> getPharmaciesSortedByZip() {
17          // needs to be implemented
18      }
19  }
```

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

```
1  class PharmacyDBTest {
2      private PharmacyDB pharmDB;
3      private Pharmacy pharm1;
4      private Pharmacy pharm2;
5      private Pharmacy pharm3;
6
7      @BeforeEach
8      void setUp() throws Exception {
9          pharmDB = new PharmacyDB();
10         pharm1=new Pharmacy("1","owner1","CVS1","addr1","", "city1"
               ,"state1", "10709", "111-1111","pharmacy");
11         pharm2=new Pharmacy("2","owner2","CVS2","addr2","22", "
               city2","state2", "22222", "222-2222","pharmacy");
12         pharm3=new Pharmacy("3","owner3","CVS3","addr3","3", "city3
               ","state3", "333333", "333-3333","pharmacy");
13         pharmDB.add(pharm2);
14         pharmDB.add(pharm1);
15         pharmDB.add(pharm3);
16     }
17 }
```

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

O.3 Software Design Question 2

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

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

```
1  public class ProductList {
2      private final int LEN = 3;
3      private Product[] products = new Product[LEN];
4      private int numProds = 0;
5
6      /**
7       * add a product if there is room
8       * @param prod
9       * @return true if the product can be added,
10      * false if there is no more space
11      */
12     public Boolean add(Product prod) {
13         if (numProds >= LEN) {
14             return false;
15         }
16         products[numProds] = prod;
17         numProds++;
18         return true;
19     }
20
21     /**
22      * return the product at position pos
23      * if there is no product at that position,
24      * return null
25      * @param pos
26      * @return a product or null if no product
27      * at that position
28      */
29     public Product getAtPos(int pos) {
30         if (pos >= numProds) {
31             return null;
32         }
```

```
33          return products[pos];
34      }
35  }
```

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

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

```
1  public interface MyIterator {
2      public Product next();
3      public boolean hasNext();
4  }
```

Here is a program that uses the iterator.

```
1  public class ProdFun {
2      public static void main(String[] args) {
3          ProductList products = new ProductList();
4          products.add(new Product("2A", "Friskies Fishalicious Cat
               Food", 12.99));
5          products.add(new Product("1B", "Fancy Feast Cat Food",
               11.88));
6          products.add(new Product("1C", "Friskies Surf N Turf Cat
               Food", 10.99));
7
8          MyIterator iter = products.getIterator();
9          while (iter.hasNext()) {
10             Product prod = iter.next();
11             System.out.println(prod.getName());
12         }
13     }
14 }
```

When run, it prints out:

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

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

O.4 Advanced Data Structures and Algorithms

This question appeared as a part of a larger exercise in a final exam for a course on advanced data structures and algorithms. The test contained a number of questions, each stating that one of the attached problem scenarios could be solved using some well-known algorithm. First we will show the question prompt, followed by the matching problem scenario (normally part of answering the question is choosing the matching problem scenario). Here is the question prompt:

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

(a) Which one?

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

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

Here is the associated problem statement:

Messy Arithmetic

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

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

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

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

Input

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

Output

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

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

Sample Input 1	**Sample Output 1**
5 0 1 2 3 9 1 2 1 -1 0 5 3 3 5 4	1 + 2 = 3 1 - -1 = 2 0 * 5 = 0 3 * 3 = 9 5 - 4 = 1
Sample Input 2	**Sample Output 2**
4 3 2 2 3 3 1 1 1 3 1 3 1	3 - 1 = 2 1 + 1 = 2 3 * 1 = 3 3 * 1 = 3

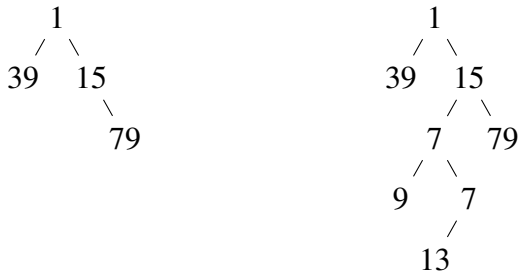
O.5 Data Structures and Algorithms 1

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

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

```
1  struct bit_node {
2    int item;
3    struct bit_node *l, *r;
4  };
5
6  typedef struct bit_node *Bit_node;
7
8  void printArray( int *a, int n ) {
9    for ( int i = 0; i < n; i++ )
10     printf( "%d ", a[i] );
11   printf( "\n" );
12 }
13
14 void f_r( Bit_node root, int *path, int len ) {
15   if (root == NULL )
16     return;
17
18   if ( root -> item % 2 ) {
19     path[len] = root -> item;
20     len++;
21   }
22
23   if ( root -> r == NULL && root -> l == NULL ) {
24     printArray( path, len );
25     return;
26   }
27
28   f_r( root -> l, path, len );
29   f_r( root -> r, path, len );
30 }
31
32 void f( Bit_node root ) {
33   Item *path = malloc( 1000 * sizeof( int ) );
34   f_r( root, path, 0 );
35 }
```

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



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

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

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

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

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

O.6 Data Structures and Algorithms 2

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

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

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

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

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

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

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

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

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

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

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

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

O.7 Data Structures and Algorithms 3

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

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

```
1  def linearSearch(table, x):
2      for i in range(len(table)):
3          if (table[i] == x):
4              return i
5      return i
6
7  def binarySearch(table, x):
8      low = 0
9      high = len(table) - 1
10
11     while (low <= high):
12         mid = (low + high) // 2
13         print(low, mid, high)
14         if (table[mid] < x):
15             low = mid + 1
16         elif (table[mid] > x):
17             high = mid - 1
18         else:
19             return mid
20     return -1
```

(a) Which of the following statements are true for `linearSearch` (L) and/or `binarySearch` (B) in case of successful search? Use one of the following five options in each case:
L = the statement is correct *only* for `linearSearch`
B = the statement is correct *only* for `binarySearch`
L&B = the statement is correct for *both* `linearSearch` and `binarySearch`
neither = not either or,
I don't know.

Each statement is worth two points as follows: correct answer +2, incorrect answer -1, empty or **I don't know** 0 points. However, you will get at least 0 points from all the questions, and the maximum is 6 x 2 p = 12 points.

i The algorithm goes through the items from smallest index to the largest.

ii The algorithm returns always the smallest index for the item x.

iii The algorithm returns all the indices that have the item x.

iv The algorithm always goes through all the items.

v The algorithm is correct only if the array is sorted in ascending order.

vi The algorithm always returns -1 at the end.

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

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

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

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

Appendix B

PLTUTOR CURRICULUM DURING 2017 EVALUATION

This appendix lists the curriculum used in PLTutor for the published evaluation. The *KnowledgeUnits* are groups of lessons on a particular topic; the notation is *KnowledgeUnit(short title shown in the help menu, short description shown in the help menu, [list of lessons])*. Each lesson is represented as an *Action(lesson's program, learning steps)*.

The learning steps provided sequential navigation for learners through a mix of showing conceptual instruction, program execution, and assessments; learning steps are represented as *LearningStep(execution step number to show, conceptual instruction, assessment question)*. The conceptual instruction of the learning step is represented as markdown to be machine-readable.

Assessment questions are represented as *Question(question prompt, distractor answers, distractor feedback)*.

```
new Curriculum(
new KnowledgeUnit("This Tutor, Learning, State, and the Stack",
            "You use the next step and previous step buttons (
                  or the arrow keys) to go through each program
                  . We show you questions only to help you
                  check that you understand the programs and
                  concepts being taught. Take your time and
                  watch how the computer works via clicking
                  next step and previous step or using the
                  arrow keys. To execute a program, the
                  computer makes an Instruction List by parsing
                   the program code, then executing each
                  instruction one by one. The State area on the
                   right shows the current instruction, and the
                   stack and namespace  before the instruction
                  is executed.",
            [
new Action (
                    new Program(`




            `
            ) /* end Program */,
                    new LearningSteps(
        /* Learning Steps */
        [
  new LearningStep(
    0,
    "## Welcome to the Programming Tutor!\nWe will help you
         learn how to read programs and understand what they
         do. But first we'll learn how to navigate and use
         this tutor.\n\n
    Press the right arrow key (or click the \"Next\" button
         above this text) to go to the next learning point.",
    undefined ),

  new LearningStep(
    0,
    "Great!\nThe text you're reading now is where we talk with
         you - it is **not** part of a program and it **does
         not** affect how the computer works.  \n\n
    You can go back to prior learning points by pressing the **
         left arrow key** (or using the \\\"Back\\\" button
         above this text). Try that now, then come back to
         here and go forward.",
    undefined ),

  new LearningStep(
    0,
    "When you learned to read and write human language, you
         started with the foundations, learning letters and
         words.\n\n
    Similarly, we'll start to learn programming by learning how
          a computer **follows** a program.\n\n
    We'll learn by looking at many different programs. Each
         program will have many learning points to go through
         .\n\n
    Press the right arrow key or click Next to continue.",
    undefined ),

  new LearningStep(
    0,
    "Each learning point highlights something to focus on. We'
         re starting by showing you explanations, which show
         up here.\n\n
    We'll use them to explain some important conceptual points
         next, then we'll look at some programs. \n\n
    Press the right arrow key or Next button to continue. We'll
          stop saying that from now on, so keep going forward
         and remember to *go back if you need to with the left
          arrow key or Back button*.",
    undefined ),

  new LearningStep(
    0,
    "## What are programs?\n                                \
         nPrograms are written in a programming language.\n\n
    **Programming languages** are extremely different from
         human languages.\n\n
    A human *language designer* makes decisions about how the
         language looks and is written (its *syntax*), and
         also how a computer behaves when it runs a program (
         called the *semantics*).\n\n
    Some of those decisions will seem confusing and mysterious
         at times, so *we'll try to understand why they were
         made*. This can help us remember how they work or
         empower us to create better languages.\n\n
    We'll learn more about the syntax and semantics of
         Javascript (which is very similar to Java, at least
         the parts we will cover here). But first we need to
         learn some concepts about computers.\n
                                        ",
    undefined ),

  new LearningStep(
    0,
    "## We will learn:\n\n
    * A computer works very differently compared to a person.\n
         \n
    * A computer only does one thing at a time.\n\n
    * To understand how the computer works, you just need to
         understand and remember many rules, where each
         individual rule is fairly simple and independent.\n\n
    * The 3 step process the computer uses to follow a program
         :\n    1. Parses Program\n    2. Creates Instruction
         List from Parsing the Program\n    3. Follows
         Instruction List\n\n
    \n* The \"state\" of the computer tells\nyou everything you
          need to know\nto understand how it works.\n\n
    * The state has 4 parts:\n  1) The list of instructions\n
         2) The instruction to execute next\n  3) The stack\n
          4) The namespace, where variables are stored\n
                ",
    undefined ),

  new LearningStep(
    0,
    "## How to Think About Computers\n\n
    Now we'll introduce some\nimportant concepts and ways to\
         nthink about computers\nand how a computer follows a
         program.\n\n
    We're doing this to help you avoid\n**misconceptions** that
          are easy\nto have about how computers follow
         programs.\nThese misconceptions sound like \n\"The
         way the computer follows a program is similar to how
         I, a person, would figure out what to do by reading
         the text of a program.\"\n\n
    These misconceptions consist of understanding how the
         computer\nappears to do something, as if the computer
          had human intelligence.",
    undefined ),

  new LearningStep(
    0,
    "It's not unreasonable to think this.\nIf you just see the
         outcomes of the computer executing a program, **
         without seeing the process**, you'd think about how
         you'd do that, then infer that the computer did it
         that way.",
    undefined ),

  new LearningStep(
    0,
    "The computer actually follows rules **blindly**, without
         thinking or taking shortcuts. This makes it different
          from how we think and follow steps. \n\n
    This difference is important because it tells you that:\n \
         n**To understand how the computer works, you just
         need to understand and remember many rules, where
         each individual rule is fairly simple and independent
         .**",
    undefined ),

  new LearningStep(
    0,
    "These rules can be hard to remember at first, because they
          don't seem too useful on their own.\n\n
    As you learn more and more of them, you'll be able to see
         how they can be used, and this will help you remember
          them.\n\n
    You'll learn each one by seeing many, many examples of it."
         ,
    undefined ),

  new LearningStep(
    0,
```

```
  "Now that we've stressed how computers follow programs
       differently than how we would, we'll describe the
       process the computer goes through to follow a program
       .\n\n
  The 3 step process the computer uses to follow a program:\
       n1. Parses Program\n2. Creates Instruction List from
       Parsing the Program\n3. Follows Instruction List\n\n
  ",
  undefined ),

new LearningStep(
  0,
  "## 1. Parses Program\n\n
  The computer runs a special program called an interpreter
       that looks at all the letters and spaces in the text
       of a program.\n\n
  We invented a verb - *parse* - to name what happens when a
       computer looks at all the letters in a program while
       it **blindly follows some rules.** ",
  undefined ),

new LearningStep(
  0,
  "## 2. Creates Instruction List\n
                                       \nWhile parsing, the
       computer follows rules to create a list of **
       instructions**.\n\n
  The instructions contain some information, but the
       information tends to be very local and small.\n\n
  For example, a single \"add\" instruction can only add two
       numbers.\n",
  undefined ),

new LearningStep(
  0,
  "## 3. Follows Instruction List\n
                                       \nThe computer stores
       the list of instructions and keeps track of which
       instruction it should execute next.\n\n
  It sets this **next instruction to execute** to refer to
       the first instruction in the list, then **executes**
       it, and goes to the next instruction.\n",
  undefined ),

new LearningStep(
  0,
  "We say a computer **executes** an instruction when it
       follows the rules for what that instruction should do
       . This is like saying it *does* the instruction.",
  undefined ),

new LearningStep(
  0,
  "So, mentally fill in the blanks below.\n\n
  The process a computer uses to follow a program is:\n1.
       Parses _____\n2. Creates _____ list from parsing the
       _____\n3. Follows _____ list",
  undefined ),

new LearningStep(
  0,
  "So, mentally fill in the blanks below.\n\n
  The process a computer uses to follow a program is:\n1.
       Parses *Program*\n2. Creates *Instruction* list from
       parsing the *Program*\n3. Follows *Instruction* list"
       ,
  undefined ),

new LearningStep(
  0,
  "Great! We're getting close to opening our first program.\n
       \n
  How do these instructions work?\n\n
  What do they do?\n\n
  Most of these instructions change the **state** of the
       computer in order to get work done.\n\n
  So what is this new concept **state**?\n",
  undefined ),

new LearningStep(
  0,
  "**State** refers to a description or record of *all* the
       different aspects that define something. It is
       everything you need to know to understand how
       something is.\n\n
  For example, we might come up with a description of an
       apple that has two pieces of state - whether it is
       fresh or rotten and whether it is sour or sweet. \n\n
  Computers have been designed by people to have *defined*
       and *exact* states. In contrast, a real world apple
       is not either fresh or rotten, it can be somewhere in
       -between.\n\n
  There are other pieces of state that define any particular
       apple, like its color.\n\n
  It's actually kind of hard to tell what particular pieces
       of state you'd need to define the state of any apple.
       ",
  undefined ),

new LearningStep(
  0,
  "In contrast, computers have been *designed* by people to
       have *clearly* defined and specific state.\n\n
  The state of the computer consists of 4 parts.\nFirst, we
       will list them, then, after a quick review, we will
       explain them and show where the tutor displays them
       for you.\n\n
  1) The list of instructions\n\n
  2) The instruction to execute next\n\n
  3) The stack\n\n
  4) The namespace, where variables are stored",
  undefined ),

new LearningStep(
  0,
  "So, to review:\n\n
  Mentally fill in the blanks.\n
                                       \nA computer works
       _____ _____ compared to a person.\n\n
                                       ",
  undefined ),

new LearningStep(
  0,
  "So, to review:\n\n
  Mentally fill in the blanks.\n\n
  A computer works very differently compared to a person. \n
                                       ",
  undefined ),

new LearningStep(
  0,
  "A computer only does ____ ____ at a time. \n
                                       ",
  undefined ),

new LearningStep(
  0,
  "A computer only does *one thing* at a time. \n
                                       ",
  undefined ),

new LearningStep(
  0,
  "To understand how the computer works, you just need to
       understand and remember many rules, where each
       individual rule is fairly ____ and _____. \n
                                       ",
  undefined ),

new LearningStep(
  0,
  "To understand how the computer works, you just need to
       understand and remember many rules, where each
       individual rule is fairly *simple* and *independent*.
        \n                             ",
  undefined ),

new LearningStep(
  0,
  "The 3 step process the computer uses to follow a program:\
       n1. _____    _____\n2. _____  _____ _____
       _____ _____ _____ _____\n3. _____  _____ _____ \
       n                               ",
  undefined ),

new LearningStep(
  0,
  "The 3 step process the computer uses to follow a program:\
       n1. Parses Program\n2. Creates Instruction List from
```

```
        Parsing the Program\n3. Follows Instruction List \n
                                        ",
    undefined ),

  new LearningStep(
    0,
    "The \"_____\" of the computer tells you everything you
        need to know\nto understand how it works. \n
                                        ",
    undefined ),

  new LearningStep(
    0,
    "The *state* of the computer tells you everything you need
        to know\nto understand how it works. \n
                                        ",
    undefined ),

  new LearningStep(
    0,
    "**Awesome!**\n\n
    Now that you have some concepts, we will get into the
        details.\n                              \nNext we'
        ll show you a simple program that doesn't really do
        much, in order to show how the tutor shows you the
        state of the computer.
                                        \n\n
    The state has 4 parts:\n1) The list of instructions\n2) The
         instruction to execute next\n3) The stack\n4) The
        namespace, where variables are stored \n
                                        ",
    undefined ),

  new LearningStep(
    0,
    "You're about to see to see a box appear under\nthe
        instruction heading to the right, in the \"State\"
        section.\nThat is not an instruction for you.\nThat's
         a description of the computer's current instruction;
         it is what the\ncomputer will do when it executes
        that instruction.\n\n
    Now, click the next program button to load your first
        program!\n",
    undefined )]
                ) /* end LearningSteps */
                ) /* end Action */,
new Action (
                new Program(`


1;




          `
          ) /* end Program */,
                new LearningSteps(
        /* Learning Steps */
        [
  new LearningStep(
    0,
    "Welcome to your first program in the tutor! This tutor
        shows the program in the middle section labeled
        Program. Right now we are looking at the program that
         is just\n                              \n1;\n
                                        ",
    undefined ),

  new LearningStep(
    0,
    "Now we will explain each part of the state and show where
        the tutor displays them for you.\n\n
    1) The list of instructions\n\n
    2) The instruction to execute next\n\n
    3) The stack\n\n
    4) The namespace, where variables are stored\n
                                        ",
    undefined ),

  new LearningStep(
    0,
    "## List of instructions\n                              \
        nThere is a timeline at the top with a mark for each
        instruction that the computer executes.\n\n
    For simple programs (particularly the ones we will start
        out with), this is the same as the list of
        instructions.\n\n
    You can click there to see what instruction was there and
        move to that point in time.\n\n
    Sometimes, we will go forward in time when you advance to a
         new learning point - the blue current instruction
        mark will move when that happens.\n
                                        ",
    undefined ),

  new LearningStep(
    0,
    "## Instruction to execute next\n
                                        \nThis orange box
        under the word \"Instruction\" is the instruction
        that will execute next.\n\n
    Right now it says \"Push 1 onto the stack.\" That describes
         what the computer will do when it executes that
        instruction.\n\n
    We also highlight the parts of the program's source code
        that are related to the instruction that is being
        executed. This is to help you understand how the
        computer follows a program.\n\n
    When a computer executes an instruction, the computer
        changes its own state.\n
                                        ",
    undefined ),

  new LearningStep(
    0,
    "## Stack\n\n
    The stack is a place you can put values (for example, a
        number is a value).\n\n
    It is called a stack because it works like a stack of cards
        . You can put something on top of the stack or take
        something off of it.\n\n
    The stack works like a short-term memory for the computer.\
        n\n
    Values go on and off the stack routinely as a program
        executes.\n                             ",
    undefined ),

  new LearningStep(
    0,
    "## Namespace\n\n
    The namespace is where variables are stored.\n\n
    This is like a table with two columns, one holds the
        variable's name, and the other holds the variable's
        value.\n",
    undefined ),

  new LearningStep(
    0,
    "## What an instruction does\n
                                        \nWhen a computer
        executes an instruction, the computer **changes its
        own state**.\n\n
    In general, the list of instructions does not change as a
        computer executes a program.\n\n
    The stack, the namespace, and the instruction to execute
        next can be changed by an instruction.\n\n
    Now, when you go to the next learning point, we will have
        the computer execute the instruction and go to the
        next instruction in the program.\n\n
    This is a simple example program so you can see the
        computer change its state and put a value on and then
         take it off of the stack. We'll get to larger
        programs later.\n\n
    \n                          ",
    undefined ),

  new LearningStep(
    1,
    "Great!\n\n
    Now you can see the value 1 is on the stack.\n\n
    From now on we will only step forward in the program's
        execution or change what we show here, not both at
        the same time. Make sure to press the right arrow key
         or Next button again after going forward to go to
        the next learning point! \n
                                        ",
    undefined ),
```

```
  new LearningStep(
    1,
    "Click the Next Program button to go to the next program,
         where we will explain more about the stack and how
         that all worked.\n                                    ",
    undefined )]
                    ) /* end LearningSteps */
                    ) /* end Action */,
new Action (
                    new Program(`

1;



2;



3;



0;



10;


11111111111111;

1938420;

230492;

020342;




            `
            ) /* end Program */,
                    new LearningSteps(
        /* Learning Steps */
        [
  new LearningStep(
    0,
    "## Expressions and the Stack\n\n
    The stack is a place you can put values.\n\n
    It is called a stack because it works like a stack of cards
         .\n\n
    You can put something on top of the stack\n(called *pushing
         *)\nor take something off of the top it\n(called *
         popping*).",
    undefined ),

  new LearningStep(
    0,
    "The stack works like a short-term memory for the computer
         .\n\n
    Values go on and off the stack routinely as a program
         executes. **Watch** how values go on and off the
         stack in these next few lines.\n\n
    Each of these lines in this program is an example of an **
         expression**. Expressions are parts of the program
         that leave a value on the stack.\n\n
    This is a simple program we'll go through just to show you
         how values go on and off the stack. \n            ",
    undefined ),

  new LearningStep(
    0,
    "Now let's go through this line. The \n\n
    1; \n\n
    will push 1 onto the stack, then remove it. The instruction
          view will describe the instruction the computer is
         doing. You can't and won't need to push something
         onto the stack.\n            ",
    undefined ),

  new LearningStep(
    1,
    "\n            ",
    undefined ),

  new LearningStep(
    1,
    "Now you can see 1 is on the stack. Let's talk about what
         the ; does in the program.\n            ",
    undefined ),

  new LearningStep(
    1,
    "The ; character signals to the computer that an expression
         is finished. When the parser sees this character, it
         creates a new instruction to remove a value from the
         stack.\n            ",
    undefined ),

  new LearningStep(
    1,
    "Why did the language designer decide for ; to do that?\n
                                          \nThey could have had
         people write \"CLEAR THE STACK\"; that would have
         made it easier for beginners to understand what the
         computer would do.\n\n
    However, people need to write that many many times in a
         program. Writing ; is much shorter, at the expense of
          needing to teach people what that means.\n
                    ",
    undefined ),

  new LearningStep(1, undefined, undefined),

  new LearningStep(2, undefined, undefined),

  new LearningStep(
    2,
    "This next line will push 2 on the stack,    then remove it
         .",
    undefined ),

  new LearningStep(2, undefined, undefined),

  new LearningStep(3, undefined, undefined),

  new LearningStep(
    4,
    "This next line will push 3 onto the stack, then remove it.
         ",
    undefined ),

  new LearningStep(4, undefined, undefined),

  new LearningStep(5, undefined, undefined),

  new LearningStep(
    6,
    "This next line will push 0 onto the stack, then remove it.
         ",
    undefined ),

  new LearningStep(6, undefined, undefined),

  new LearningStep(7, undefined, undefined),

  new LearningStep(
    8,
    "This next line will push 10 onto the stack, then remove it
         .\n\n
    Note that it still just takes *one* instruction, no matter
         how long the number is.",
    undefined ),

  new LearningStep(8, undefined, undefined),

  new LearningStep(9, undefined, undefined),

  new LearningStep(
    10,
    "Note that it still just takes\none instruction no matter\
         nhow long the number is.",
    undefined ),

  new LearningStep(10, undefined, undefined),

  new LearningStep(11, undefined, undefined),

  new LearningStep(12, undefined, undefined),

  new LearningStep(13, undefined, undefined),
```

```
  new LearningStep(14, undefined, undefined),

  new LearningStep(15, undefined, undefined),

  new LearningStep(16, undefined, undefined),

  new LearningStep(17, undefined, undefined),

  new LearningStep(
    17,
    "Click Next Program to continue. ",
    undefined )]
                    ) /* end LearningSteps */
                    ) /* end Action */,
new Action (
                    new Program(`


1;
```

```
            `
            ) /* end Program */,
                    new LearningSteps(
        /* Learning Steps */
        [
  new LearningStep(
    0,
    "## Questions\nTo help you learn, we have questions for you
        , which will cover up some parts of the computer's **
        state** and ask you what should go there.\n
                                ",
    undefined ),

  new LearningStep(
    0,
    "If you feel you're understanding\nwhat's happening, let
        the tutor know by marking 'I'm Sure' after your
        choice.\n\n
    If you feel less confident,\nmark the other choice. This
        helps us understand you and make the tutor better in
        the future.\n                          ",
    undefined ),

  new LearningStep(
    0,
    "\n                          \n                          ",
    undefined ),

  new LearningStep(1, undefined, undefined),

  new LearningStep(
    1,
    undefined,

    new Question(
      "What value is on the stack?",
      [[2,5,7],true,"An African Swallow"],
      [] ) ),

  new LearningStep(
    1,
    "If you're not sure of something, **try navigating
        backwards or forwards** - this can help you figure
        out what is happening. \n                          ",
    undefined ),

  new LearningStep(
    1,
    "You can change your answer at any time; to advance, you'll
         need to match what the computer does for each
        question.\n                          ",
    undefined ),

  new LearningStep(
    1,
    "Now, you could spend the next few hours randomly clicking
        on answers to advance \"faster\" but you will not
        learn what's going on (trust us, we've seen some
        people do that, and all of them told us they
        regretted it later).\n                          ",
    undefined ),

  new LearningStep(
    1,
    "Now, on to the next program!\n                          ",
    undefined )]
                    ) /* end LearningSteps */
                    ) /* end Action */,
new Action (
                    new Program(`


1;



2;



3;
```

```
            `
            ) /* end Program */,
                    new LearningSteps(
        /* Learning Steps */
        [
  new LearningStep(
    0,
    "## Stepping\n\n
    The learning points will walk you through each program to
        observe what the computer does as it executes each
        instruction.\n\n
    This is called **stepping** through\nor watching a program
        .\n\n
    The tutor will work with you;\nit will ask questions at
        times\nwhen you probably won't know what's happening
        .\nThat's just part of the learning style for this
        environment. \n\n
    Now, use the right and left arrow keys to step forward **
        and backwards** to walk through the sequence of
        instructions, while paying attention to what is
        happening at each step.\n\n
    We'll spend a good deal of time on the basics to create a
        strong foundation.\n",
    undefined ),

  new LearningStep(0, undefined, undefined),

  new LearningStep(1, undefined, undefined),

  new LearningStep(2, undefined, undefined),

  new LearningStep(3, undefined, undefined),

  new LearningStep(4, undefined, undefined),

  new LearningStep(5, undefined, undefined),

  new LearningStep(
    5,
    undefined,

    new Question(
      "What value is on the stack?",
      [1,2,4],
      [] ) ),

  new LearningStep(
    5,
    "You can go faster when you feel you understand, but *go
        back if questions show you don't fully understand yet
        .*",
    undefined ),

  new LearningStep(
    5,
    "Click the next program button to continue!",
    undefined )]
                    ) /* end LearningSteps */
```

```
                    ) /* end Action */,
new Action (
                    new Program(`

1;

2;


3;

5;

0;

20;

3000;

394320;

2934023;



            `
            ) /* end Program */,
                    new LearningSteps(
          /* Learning Steps */
          [
  new LearningStep(
    0,
    "Time for some questions",
    undefined ),

  new LearningStep(0, undefined, undefined),

  new LearningStep(1, undefined, undefined),

  new LearningStep(
    1,
    undefined,

    new Question(
      "What value is on the stack?",
      [0,2,4],
      [] ) ),

  new LearningStep(2, undefined, undefined),

  new LearningStep(3, undefined, undefined),

  new LearningStep(
    3,
    undefined,

    new Question(
      "What value is on the stack?",
      [1,3,4],
      [] ) ),

  new LearningStep(4, undefined, undefined),

  new LearningStep(5, undefined, undefined),

  new LearningStep(
    5,
    undefined,

    new Question(
      "What value is on the stack?",
      [1,2,4],
      [] ) ),

  new LearningStep(6, undefined, undefined),

  new LearningStep(7, undefined, undefined),

  new LearningStep(
    7,
    undefined,

    new Question(
      "What value is on the stack?",
      [2,3,4],
      [] ) ),

  new LearningStep(8, undefined, undefined),

  new LearningStep(9, undefined, undefined),

  new LearningStep(
    9,
    undefined,

    new Question(
      "What value is on the stack?",
      [1,2,4],
      [] ) ),

  new LearningStep(10, undefined, undefined),

  new LearningStep(11, undefined, undefined),

  new LearningStep(
    11,
    undefined,

    new Question(
      "What value is on the stack?",
      [10,30,40],
      [] ) ),

  new LearningStep(12, undefined, undefined),

  new LearningStep(13, undefined, undefined),

  new LearningStep(
    13,
    undefined,

    new Question(
      "What value is on the stack?",
      [1000,20,3001],
      [] ) ),

  new LearningStep(14, undefined, undefined),

  new LearningStep(15, undefined, undefined),

  new LearningStep(
    15,
    undefined,

    new Question(
      "What value is on the stack?",
      [39,43,20],
      [] ) ),

  new LearningStep(16, undefined, undefined),

  new LearningStep(17, undefined, undefined),

  new LearningStep(
    17,
    undefined,

    new Question(
      "What value is on the stack?",
      [29,340,23],
      [] ) ),

  new LearningStep(
    17,
    "## Resuming where you left off\n\n
    You can click on the word **Help** at the top to open the
         Learning Plan if you want to **go back** to a prior
         program. You can also use this to open any program in
          case you take a break and **close this window** (or
         have some issue and close it unintentionally). Once
         you click on the program you want, click \"Close\" at
          the top and it will close.\n\n
    There's a summary up there of the section you're on as well
         . Try clicking on the word Help now then click Close.
         ",
    undefined ),

  new LearningStep(
```

```
    17,
    "## Onwards!\n\n
    Click Next Program!",
    undefined )]
                    ) /* end LearningSteps */
                    ) /* end Action */]),
new KnowledgeUnit("Variables",
            "A variable is like a box. We tell the computer to
                  make a place to put something. We give that
                  place a name, so we can remember it and what
                  we want to use it for. A variable gets a
                  place in the Namespace. We can give it a
                  value to start with. The value can change
                  later.",
            [
new Action (
                    new Program(`
var box = 1;



var another_box = 2;



var a_third_box = 0;



var box4 = 10;


var ICanNameVariables = 20;

var foo = 10;

var a = 2000;

var b = 15;

var c = 35;

            `
            ) /* end Program */,
                    new LearningSteps(
        /* Learning Steps */
        [
  new LearningStep(
    0,
    "## Variables\n\n
    If you tried to write a large program all in a single
          expression,\nit would get pretty hard to understand.\
          nSo, language designers have created a feature called
           **variables**.\n\n
    A variable is like a box.\nWe tell the computer\nto make a
          place to put something\nand give that place a name,\
          nso we can remember it\nand remember what we want to
          use it for.\n\n
    It gets a place in the *namespace*,\nwhich you can see
          shown underneath the stack.\nWe can give it a value
          to start with\nafter the equals sign (=).\n\n
    The Namespace is like a table with two columns;\nthe first
          column has the variable name,\nand the other has the
          value stored\nin that variable.",
    undefined ),

  new LearningStep(
    0,
    "Let's step through the program.\n\n
    Pay attention to how the computer uses the value on the
          stack\nwhen it executes the create variable
          instruction.\n\n
    In this line, we'll name this variable box and store 1 in
          it.",
    undefined ),

  new LearningStep(0, undefined, undefined),

  new LearningStep(1, undefined, undefined),

  new LearningStep(2, undefined, undefined),

  new LearningStep(3, undefined, undefined),

  new LearningStep(4, undefined, undefined),

new LearningStep(
  4,
  "You may have expected the computer to act differently from
        how it did.\n\n
  If you're feeling confused\nbecause of that, it is just
        because the computer works differently from people.
        Watch and see\nhow the computer follows the program.
        It usually will *not* work the way you think it will
        .\n",
  undefined ),

new LearningStep(
  4,
  "Watch how the **namespace**\nand **stack** change, and use
        the left and right arrow keys to go back and forward
        to watch and re-watch as needed.",
  undefined ),

new LearningStep(5, undefined, undefined),

new LearningStep(6, undefined, undefined),

new LearningStep(7, undefined, undefined),

new LearningStep(8, undefined, undefined),

new LearningStep(9, undefined, undefined),

new LearningStep(
  10,
  "Look in the namespace and find the variable named box.\
        nSee that the variable named box still holds the
        value 1.",
  undefined ),

new LearningStep(10, undefined, undefined),

new LearningStep(11, undefined, undefined),

new LearningStep(12, undefined, undefined),

new LearningStep(13, undefined, undefined),

new LearningStep(14, undefined, undefined),

new LearningStep(
  15,
  "Why do the values get taken off the stack?\nIf the values
        stayed on the stack, it would get really big.\n\n
  The stack is like a *temporary place*. **Variables** are
        where we put values we want to\nuse later.",
  undefined ),

new LearningStep(15, undefined, undefined),

new LearningStep(16, undefined, undefined),

new LearningStep(17, undefined, undefined),

new LearningStep(18, undefined, undefined),

new LearningStep(19, undefined, undefined),

new LearningStep(
  20,
  "Variables can be named almost anything.",
  undefined ),

new LearningStep(21, undefined, undefined),

new LearningStep(22, undefined, undefined),

new LearningStep(23, undefined, undefined),

new LearningStep(24, undefined, undefined),

new LearningStep(25, undefined, undefined),

new LearningStep(26, undefined, undefined),

new LearningStep(27, undefined, undefined),

new LearningStep(28, undefined, undefined),
```

```
new LearningStep(
  28,
  undefined,

  new Question(
    "What is the value of the foo variable after this
         instruction?",
    [29,19,15],
    [] ) ),

new LearningStep(29, undefined, undefined),

new LearningStep(30, undefined, undefined),

new LearningStep(31, undefined, undefined),

new LearningStep(32, undefined, undefined),

new LearningStep(33, undefined, undefined),

new LearningStep(
  33,
  undefined,

  new Question(
    "What is the value of the a variable after this
         instruction?",
    [1996,2016,2008],
    [] ) ),

new LearningStep(34, undefined, undefined),

new LearningStep(35, undefined, undefined),

new LearningStep(36, undefined, undefined),

new LearningStep(37, undefined, undefined),

new LearningStep(38, undefined, undefined),

new LearningStep(
  38,
  undefined,

  new Question(
    "What is the value of the b variable after this
         instruction?",
    [10,20,35],
    [] ) ),

new LearningStep(39, undefined, undefined),

new LearningStep(
  39,
  "You **do not** have to read the description under the **
       instruction** (in the State area) for every step all
       the time. Read it when you are learning what that
       kind of instruction does. You can also read it to
       check your understanding or if you are confused.",
  undefined ),

new LearningStep(
  39,
  "Try using the right and left arrow keys to go forward and
       back instead of clicking with the mouse. For most
       people this is better after they try it for a while."
       ,
  undefined ),

new LearningStep(40, undefined, undefined),

new LearningStep(41, undefined, undefined),

new LearningStep(42, undefined, undefined),

new LearningStep(43, undefined, undefined),

new LearningStep(
  43,
  undefined,

  new Question(
    "What is the value of the c variable after this
         instruction?",
    [15,30,5],
    [] ) ),

  new LearningStep(
    44,
    "Great, onward to the next program!",
    undefined )]
                    ) /* end LearningSteps */
                    ) /* end Action */,
new Action (
                    new Program(`


var number_of_kindergardeners = 200;
var number_of_staff = 30;
var number_of_teachers = 50;
var number_of_principles = 1;



var num_kindergardeners = 200;
var n_kindergardeners = 200;


var nK = 200;



var b9 = 20;
var c927j = 100;









          `
          ) /* end Program */,
                  new LearningSteps(
        /* Learning Steps */
        [
  new LearningStep(
    0,
    "## Choosing variable names\n\n
    Programmers usually choose\nvariable names that relate to
         whatever they are trying to do. \n\n
    For example,\nin a program that calculates\nthe total
         number of people in a school,\nit might start out
         with putting\nthe class sizes in different variables.
         ",
    undefined ),

  new LearningStep(0, undefined, undefined),

  new LearningStep(1, undefined, undefined),

  new LearningStep(2, undefined, undefined),

  new LearningStep(3, undefined, undefined),

  new LearningStep(
    3,
    undefined,

    new Question(
      "What value does number_of_kindergardeners have?",
      [20,0,2000],
      [] ) ),

  new LearningStep(4, undefined, undefined),

  new LearningStep(5, undefined, undefined),

  new LearningStep(6, undefined, undefined),

  new LearningStep(7, undefined, undefined),

  new LearningStep(8, undefined, undefined),

  new LearningStep(
    8,
    undefined,

    new Question(
      "What value does number_of_staff have?",
```

```
    [40,10,3],
    [] ) ),

new LearningStep(9, undefined, undefined),

new LearningStep(10, undefined, undefined),

new LearningStep(11, undefined, undefined),

new LearningStep(12, undefined, undefined),

new LearningStep(13, undefined, undefined),

new LearningStep(14, undefined, undefined),

new LearningStep(15, undefined, undefined),

new LearningStep(
  15,
  undefined,

  new Question(
    "What value does the variable number_of_teachers have? ",
    [10,0,5],
    ["","",""] ) ),

new LearningStep(16, undefined, undefined),

new LearningStep(17, undefined, undefined),

new LearningStep(18, undefined, undefined),

new LearningStep(19, undefined, undefined),

new LearningStep(
  19,
  "When you get to a question, you won't be able to advance
       until answering it. So you **do not need to worry**
       about accidentally missing a question.",
  undefined ),

new LearningStep(
  19,
  "Whenever an explanation appears here, we **pause** you for
       about a second so you can't step forward. This helps
       prevent missing an explanation step.\n\n
   So if you are stepping and you stop going forward, that
       may be what is happening.",
  undefined ),

new LearningStep(
  20,
  "There would probably be\nmore variables for \nother grades
       .\n\n
  \nProgrammers often *abbreviate words* in order\nto type
       less. \n\n
  For example, var is an abbreviation for variable that the
       language designer used.\n\n
  For another example, here's how someone might abbreviate
       number_of_kindergardeners to num_kindergardeners.",
  undefined ),

new LearningStep(20, undefined, undefined),

new LearningStep(21, undefined, undefined),

new LearningStep(22, undefined, undefined),

new LearningStep(23, undefined, undefined),

new LearningStep(24, undefined, undefined),

new LearningStep(25, undefined, undefined),

new LearningStep(26, undefined, undefined),

new LearningStep(27, undefined, undefined),

new LearningStep(28, undefined, undefined),

new LearningStep(29, undefined, undefined),

new LearningStep(
  30,
  "Sometimes programmers abbreviate *a little too much*.\
       nAbbreviating your variable names\nmakes it hard for
       people to read\nand understand your code.\nImagine if
        someone wrote this variable\nfor the number of
       kindergardeners.",
  undefined ),

new LearningStep(30, undefined, undefined),

new LearningStep(31, undefined, undefined),

new LearningStep(32, undefined, undefined),

new LearningStep(33, undefined, undefined),

new LearningStep(34, undefined, undefined),

new LearningStep(
  35,
  "It's easy for the computer to make\na variable like that,
       though it's \nhard for people to understand what it
       is going to be used for.\n\n
  We aren't going to use many\nunderstandable names for\
       nvariables in programs\nin the rest of our example
       programs,\nbecause **the computer doesn't need them
       to be understandable.** \n\n
  To the computer,\na variable name is just a sequence of
       individual characters\nand it doesn't have any
       meaning.",
  undefined ),

new LearningStep(35, undefined, undefined),

new LearningStep(36, undefined, undefined),

new LearningStep(37, undefined, undefined),

new LearningStep(38, undefined, undefined),

new LearningStep(39, undefined, undefined),

new LearningStep(40, undefined, undefined),

new LearningStep(41, undefined, undefined),

new LearningStep(42, undefined, undefined),

new LearningStep(43, undefined, undefined),

new LearningStep(44, undefined, undefined),

new LearningStep(
  44,
  undefined,

  new Question(
    "Now that we're at the end of the program, what value
         does number_of_principles have?",
    [100,0,5],
    ["","",""] ) ),

new LearningStep(
  44,
  "Onwards!",
  undefined )]
                ) /* end LearningSteps */
                ) /* end Action */]),
new KnowledgeUnit("Variables - details and changing values",
        "A variable is like a box. We tell the computer to
             make a place to put something. We give that
             place a name, so we can remember it and what
             we want to use it for. A variable gets a
             place in the Namespace. We can give it a
             value to start with. The value can change
             later.",
        [
new Action (
                new Program(`

var do_not_put_10_into_this = 10;

var this_box_never_equals_2 = 2;


var ignore_this_var = 10000;
```

```
var really_dont_make_this_var = 5;
```

```
          `
          ) /* end Program */,
                  new LearningSteps(
       /* Learning Steps */
       [
  new LearningStep(
    0,
    "The computer does **not** use context or the variable's
         name\nto change how the program executes; it just
         follows\nthe instructions that the parser made from
         the program.",
    undefined ),

  new LearningStep(0, undefined, undefined),

  new LearningStep(1, undefined, undefined),

  new LearningStep(2, undefined, undefined),

  new LearningStep(3, undefined, undefined),

  new LearningStep(4, undefined, undefined),

  new LearningStep(5, undefined, undefined),

  new LearningStep(6, undefined, undefined),

  new LearningStep(7, undefined, undefined),

  new LearningStep(8, undefined, undefined),

  new LearningStep(
    8,
    undefined,

    new Question(
      "What is the variable's value after this instruction
           executes?",
      [0,3,4],
      [] ) ),

  new LearningStep(9, undefined, undefined),

  new LearningStep(
    10,
    "Just to be sure, double check that what I said is true.
         Look at the *namespace* to see the values of the
         variables.",
    undefined ),

  new LearningStep(10, undefined, undefined),

  new LearningStep(11, undefined, undefined),

  new LearningStep(12, undefined, undefined),

  new LearningStep(13, undefined, undefined),

  new LearningStep(14, undefined, undefined),

  new LearningStep(15, undefined, undefined),

  new LearningStep(16, undefined, undefined),

  new LearningStep(17, undefined, undefined),

  new LearningStep(18, undefined, undefined),

  new LearningStep(
    18,
    undefined,

    new Question(
      "What is the variable's value after this instruction
           executes?",
      [null,0,-1],
      [] ) ),

  new LearningStep(
    19,
    "Let's go to the next program!",
    undefined )]
                  ) /* end LearningSteps */
                  ) /* end Action */,
new Action (
                  new Program(`

var name1 = 1;

name1 = 10;
```

```
          `
          ) /* end Program */,
                  new LearningSteps(
       /* Learning Steps */
       [
  new LearningStep(
    0,
    "## Variable values can change\n\n
    Watch the namespace to see the values of variables as the
         program executes and exactly when they change.",
    undefined ),

  new LearningStep(0, undefined, undefined),

  new LearningStep(1, undefined, undefined),

  new LearningStep(2, undefined, undefined),

  new LearningStep(3, undefined, undefined),

  new LearningStep(4, undefined, undefined),

  new LearningStep(
    4,
    undefined,

    new Question(
      "What is the name1 variable's value?",
      [2,0,"name1"],
      [] ) ),

  new LearningStep(
    4,
    "Now watch the next steps closely.",
    undefined ),

  new LearningStep(5, undefined, undefined),

  new LearningStep(6, undefined, undefined),

  new LearningStep(7, undefined, undefined),

  new LearningStep(
    7,
    undefined,

    new Question(
      "What is name1's value now?",
      [1,11,2],
      ["It's value used to be 1 but the instructions for this
           line of code changed it.","The equal sign sets it
           to the value on the stack; it does NOT add them
           together.",""] ) )]
                  ) /* end LearningSteps */
                  ) /* end Action */,
new Action (
                  new Program(`

var name1 = 1;

name1 = 2;
name1 = 3;
```

```
name1 = 1;
name1 = 50;
```

```
          `
          ) /* end Program */,
                    new LearningSteps(
        /* Learning Steps */
        [
  new LearningStep(
    0,
    "Since this is so important and used so frequently in
         programs, we'll do some more examples. Watch the
         namespace to see the values of variables as the
         program executes and exactly when they change.",
    undefined ),

  new LearningStep(0, undefined, undefined),

  new LearningStep(1, undefined, undefined),

  new LearningStep(2, undefined, undefined),

  new LearningStep(3, undefined, undefined),

  new LearningStep(
    3,
    undefined,

    new Question(
      "What is name1's value after this instruction executes?",
      [2,3,0],
      [] ) ),

  new LearningStep(4, undefined, undefined),

  new LearningStep(5, undefined, undefined),

  new LearningStep(6, undefined, undefined),

  new LearningStep(
    6,
    undefined,

    new Question(
      "What is name1's value after this instruction executes?",
      ["2;",4,1],
      ["The parser turns the ; into a different instruction, so
           it doesn't get put in name1.","","name1's initial
         value was 1 but this instruction changes it to 2.
         Before this step its value is 1."] ) ),

  new LearningStep(7, undefined, undefined),

  new LearningStep(8, undefined, undefined),

  new LearningStep(9, undefined, undefined),

  new LearningStep(
    9,
    undefined,

    new Question(
      "What is name1's value after this instruction executes?",
      [2,1,50],
      ["name1's value was 2 but this instruction changes it to
           3. We're asking what it's value is AFTER this
         instruction executes.","The value of name1 changes
         at multiple parts of the program; you can't just
         look at where it first appears to find the value.",
         "Right now the last line with name1 = 50 has not
         executed yet. The orange highlight in the code
         shows you what time step you are on, as well as the
          blue box on the timeline at the top of the screen.
         "] ) ),

  new LearningStep(10, undefined, undefined),

  new LearningStep(11, undefined, undefined),

  new LearningStep(12, undefined, undefined),

  new LearningStep(
    12,
    undefined,

    new Question(
      "What is name1's value BEFORE this instruction executes?"
          ,
      [2,1,50],
      ["name1's value was 2 several steps ago but it got
           changed on the previous line. The highlight in the
           code shows you what time step you are on, as well
           as the blue box on the timeline at the top of the
           screen.","name1 will have the value of 1 AFTER this
            instruction executes. Right now you are at the
           time step before this instruction executes.","Right
            now the last line with name1 = 50 has not executed
            yet. The highlight in the code shows you what time
            step you are on, as well as the blue box on the
           timeline at the top of the screen."] ) ),

  new LearningStep(13, undefined, undefined),

  new LearningStep(14, undefined, undefined),

  new LearningStep(15, undefined, undefined),

  new LearningStep(
    15,
    undefined,

    new Question(
      "What is name1's value after this instruction executes?",
      [3,2,1],
      ["name1's value was 3 several steps ago but it got
           changed. The highlight in the code shows you what
           time step you are on, as well as the blue box on
           the timeline at the top of the screen. Look at the
           text in the instrution box to see what this
           instrution does.","name1's value was 2 several
           steps ago but it got changed. The highlight in the
           code shows you what time step you are on, as well
           as the blue box on the timeline at the top of the
           screen. Look at the text in the instrution box to
           see what this instrution does.","The value of name1
            changes at multiple parts of the program; you can'
           t just look at where it first appears to find the
           value."] ) ),

  new LearningStep(
    16,
    "Great! Forward!",
    undefined )]
                    ) /* end LearningSteps */
                    ) /* end Action */,
new Action (
                    new Program(`


var a = 1;
var b = 2;
var c = 3;

a;
b;
c;


a;
1;

a;
1;

b;
2;

c;
3;

c;
3;

b;
```

```
          `
          ) /* end Program */,
                    new LearningSteps(
      /* Learning Steps */
      [
new LearningStep(
  0,
  "## Using Variables\n\n
  Great! You're doing the work needed to understand\
       nvariables.\nSo far, you've only seen programs that\
       nput a number on the right hand side of the equal
       sign.\n\n
  Let's start using variables!\nWhen the computer sees a
       variable name that is not\nimmediately to the left of
        an equal sign, it **looks\nup its value** and **
       pushes it onto the stack**.\n\n
  We'll go through some basic examples first.",
  undefined ),

new LearningStep(0, undefined, undefined),

new LearningStep(1, undefined, undefined),

new LearningStep(2, undefined, undefined),

new LearningStep(3, undefined, undefined),

new LearningStep(4, undefined, undefined),

new LearningStep(5, undefined, undefined),

new LearningStep(6, undefined, undefined),

new LearningStep(7, undefined, undefined),

new LearningStep(8, undefined, undefined),

new LearningStep(9, undefined, undefined),

new LearningStep(10, undefined, undefined),

new LearningStep(11, undefined, undefined),

new LearningStep(12, undefined, undefined),

new LearningStep(13, undefined, undefined),

new LearningStep(14, undefined, undefined),

new LearningStep(
  14,
  "Now, a's value is about to get put on the stack in the
       next line.",
  undefined ),

new LearningStep(15, undefined, undefined),

new LearningStep(16, undefined, undefined),

new LearningStep(
  16,
  "There it is! 1 was in a, now a value of 1 is on the stack
       at this moment.",
  undefined ),

new LearningStep(17, undefined, undefined),

new LearningStep(18, undefined, undefined),

new LearningStep(19, undefined, undefined),

new LearningStep(20, undefined, undefined),

new LearningStep(
  21,
  "Look at how these leave the same value on the stack before
       the ;",
  undefined ),

new LearningStep(21, undefined, undefined),

new LearningStep(22, undefined, undefined),

new LearningStep(23, undefined, undefined),

new LearningStep(24, undefined, undefined),

new LearningStep(25, undefined, undefined),

new LearningStep(26, undefined, undefined),

new LearningStep(27, undefined, undefined),

new LearningStep(28, undefined, undefined),

new LearningStep(29, undefined, undefined),

new LearningStep(30, undefined, undefined),

new LearningStep(
  30,
  undefined,

  new Question(
    "What is on the stack?",
    [3,1,0],
    ["bs value was set in the second line of the program.
         you can look back to find the closest appearance of
          b = to figure out its value.","bs value was
         set in the second line of the program. you can look
          back to find the closest appearance of b = to
         figure out its value.","bs value was set in
         the second line of the program. you can look back
         to find the closest appearance of b = to figure out
          its value."] ) ),

new LearningStep(31, undefined, undefined),

new LearningStep(32, undefined, undefined),

new LearningStep(33, undefined, undefined),

new LearningStep(34, undefined, undefined),

new LearningStep(
  34,
  undefined,

  new Question(
    "What is on the stack?",
    [2,1,0],
    ["cs value was set in the second line of the program.
         you can look back to find the closest appearance of
          c = to figure out its value.","cs value was
         set in the second line of the program. you can look
          back to find the closest appearance of c = to
         figure out its value.","cs value was set in
         the second line of the program. you can look back
         to find the closest appearance of c = to figure out
          its value."] ) ),

new LearningStep(35, undefined, undefined),

new LearningStep(36, undefined, undefined),

new LearningStep(37, undefined, undefined),

new LearningStep(38, undefined, undefined),

new LearningStep(39, undefined, undefined),

new LearningStep(40, undefined, undefined),

new LearningStep(41, undefined, undefined),

new LearningStep(
  42,
  "Great! Onward!",
  undefined )]
                    ) /* end LearningSteps */
                    ) /* end Action */,
new Action (
                    new Program(`
var name1 = 5029;
var name2 = 10;
```

```
name1 = name2;



var set_this_to_10= 10;
set_this_to_10 = 5;
set_this_to_10 = name1;
set_this_to_10 = name2;




            `
            ) /* end Program */,
                    new LearningSteps(
        /* Learning Steps */
        [
  new LearningStep(
    0,
    "## Practice\n\n
    Let's practice changing variable values.",
    undefined ),

  new LearningStep(1, undefined, undefined),

  new LearningStep(2, undefined, undefined),

  new LearningStep(3, undefined, undefined),

  new LearningStep(4, undefined, undefined),

  new LearningStep(5, undefined, undefined),

  new LearningStep(6, undefined, undefined),

  new LearningStep(7, undefined, undefined),

  new LearningStep(8, undefined, undefined),

  new LearningStep(9, undefined, undefined),

  new LearningStep(10, undefined, undefined),

  new LearningStep(11, undefined, undefined),

  new LearningStep(12, undefined, undefined),

  new LearningStep(
    13,
    "The computer doesn't use context or the variable's name to
          change how the program executes.\n\n
    It just follows\nthe program as written \naccording to the
         rules of the language.",
    undefined ),

  new LearningStep(13, undefined, undefined),

  new LearningStep(14, undefined, undefined),

  new LearningStep(15, undefined, undefined),

  new LearningStep(16, undefined, undefined),

  new LearningStep(17, undefined, undefined),

  new LearningStep(
    17,
    undefined,

    new Question(
      "What is set_this_to_10's value right now?",
      [0,5,1],
      [] ) ),

  new LearningStep(18, undefined, undefined),

  new LearningStep(19, undefined, undefined),

  new LearningStep(20, undefined, undefined),

  new LearningStep(
    20,
    undefined,

    new Question(
      "What is set_this_to_10's value?",
      [0,10,1],
      ["Even though the variable name is set_this_to_10 the
           computer follows this code according to rules. It
           pushes the value to the right of the equals sign
           onto the stack, then assigns that value to
           set_this_to_10","Even though the variable name is
           set_this_to_10 the computer follows this code
           according to rules. It pushes the value to the
           right of the equals sign onto the stack, then
           assigns that value to set_this_to_10","Even though
           the variable name is set_this_to_10 the computer
           follows this code according to rules. It pushes the
            value to the right of the equals sign onto the
           stack, then assigns that value to set_this_to_10"]
           ) ),

  new LearningStep(21, undefined, undefined),

  new LearningStep(22, undefined, undefined),

  new LearningStep(23, undefined, undefined),

  new LearningStep(
    23,
    undefined,

    new Question(
      "What is set_this_to_10's value?",
      [0,5,5029],
      ["","That's the value it used to have, but it changed.","
           You cant just look to where the variable was
           created; its value can change, so you have to
           follow the whole program and all the steps from the
            beginning. Does name1s value change anywhere?"]
           ) ),

  new LearningStep(24, undefined, undefined),

  new LearningStep(25, undefined, undefined),

  new LearningStep(
    26,
    "The computer follows the program as written. No ordinary
         programmer would write programs like this, because
         they also want people to easily understand them. Now,
          next program!",
    undefined )]
                    ) /* end LearningSteps */
                    ) /* end Action */]),
new KnowledgeUnit("Boolean Values (True False)",
          "There are two special 'boolean' values - 'true'
                and 'false'. They work just like number
                values - they go on and off the stack and can
                 be stored in variables and arrays. The
                computer uses them to make 'decisions' in if
                statements and loops. True means 'yes' and
                false means 'no'.",
           [
new Action (
                    new Program(`

true;
false;

true;
false;


var a = true;
var b = false;
a;
b;
b = true;
a = false;
a;
b;
a = b;
a;
b;
b = a;
```

```
a;
b;


            `
            ) /* end Program */,
                    new LearningSteps(
        /* Learning Steps */
        [
  new LearningStep(
    0,
    "## Boolean Values\n\n
    Great job with variables!\n\n
    We've learned about number values so far. **Boolean**
         values work the same as numbers.\n\n
    The word **boolean** is an adjective.\n\n
    When we say boolean values, we\nmean values that are one of
         \ntwo special values: *true* or *false*.\n\n
    ",
    undefined ),

  new LearningStep(
    0,
    "They go on and off the stack\njust like numbers.\n\n
    Let's step through and watch\nthem go on and off\nthe stack
        .\n",
    undefined ),

  new LearningStep(0, undefined, undefined),

  new LearningStep(1, undefined, undefined),

  new LearningStep(2, undefined, undefined),

  new LearningStep(3, undefined, undefined),

  new LearningStep(
    4,
    "",
    undefined ),

  new LearningStep(
    5,
    "",
    undefined ),

  new LearningStep(
    5,
    "What value is on the stack?",

    new Question(
      "What value is on the stack?",
      [false,10,-10],
      ["","The answer is a boolean, not a number.","The answer
           is a boolean, not a number."] ) ),

  new LearningStep(
    6,
    "",
    undefined ),

  new LearningStep(
    7,
    "",
    undefined ),

  new LearningStep(
    7,
    "What value is on the stack?",

    new Question(
      "What value is on the stack?",
      [true,-10,10],
      ["","The answer is a boolean, not a number.","The answer
           is a boolean, not a number."] ) ),

  new LearningStep(
    8,
    "**Boolean** values can be stored in variables, just like
         numbers.",
    undefined ),

  new LearningStep(8, undefined, undefined),

  new LearningStep(9, undefined, undefined),

  new LearningStep(10, undefined, undefined),

  new LearningStep(11, undefined, undefined),

  new LearningStep(12, undefined, undefined),

  new LearningStep(13, undefined, undefined),

  new LearningStep(14, undefined, undefined),

  new LearningStep(15, undefined, undefined),

  new LearningStep(16, undefined, undefined),

  new LearningStep(17, undefined, undefined),

  new LearningStep(18, undefined, undefined),

  new LearningStep(
    18,
    undefined,

    new Question(
      "What value is on the stack after this instruction
           executes?",
      [false,10,-10],
      ["","The answer is not a number, just a boolean value.","
           The answer is not a number, just a boolean value."]
           ) ),

  new LearningStep(19, undefined, undefined),

  new LearningStep(20, undefined, undefined),

  new LearningStep(21, undefined, undefined),

  new LearningStep(
    21,
    undefined,

    new Question(
      "What is the value of b?",
      [true,-10,10],
      ["","The answer is not a number, just a boolean value.","
           The answer is not a number, just a boolean value."]
           ) ),

  new LearningStep(22, undefined, undefined),

  new LearningStep(23, undefined, undefined),

  new LearningStep(24, undefined, undefined),

  new LearningStep(25, undefined, undefined),

  new LearningStep(26, undefined, undefined),

  new LearningStep(
    26,
    undefined,

    new Question(
      "What is the value of a before this instruction executes?
           ",
      [false,10,-10],
      ["This instruction hasn't executed yet, so a still has
           its previous value.","The answer is not a number,
           just a boolean value.","The answer is not a number,
           just a boolean value."] ) ),

  new LearningStep(27, undefined, undefined),

  new LearningStep(28, undefined, undefined),

  new LearningStep(29, undefined, undefined),

  new LearningStep(30, undefined, undefined),

  new LearningStep(
    30,
    undefined,

    new Question(
      "What is the value of a (not the b variable)?",
```

```
      [true,10,-10],
      ["The a variable started out as true but it was changed.
            You can't just look at where a was declared; you
            look at the program up to this point.","The answer
            is not a number, just a boolean value.","The answer
             is not a number, just a boolean value."] ) ),

  new LearningStep(31, undefined, undefined),

  new LearningStep(
    31,
    undefined,

    new Question(
      "What is the value of b?",
      [false,10,-10],
      ["","The answer is not a number, just a boolean value.","
            The answer is not a number, just a boolean value."]
            ) ),

  new LearningStep(32, undefined, undefined),

  new LearningStep(33, undefined, undefined),

  new LearningStep(34, undefined, undefined),

  new LearningStep(35, undefined, undefined),

  new LearningStep(36, undefined, undefined),

  new LearningStep(37, undefined, undefined),

  new LearningStep(
    37,
    undefined,

    new Question(
      "What is the value of a?",
      [false,10,-10],
      ["","The answer is not a number, just a boolean value.","
            The answer is not a number, just a boolean value."]
            ) ),

  new LearningStep(38, undefined, undefined),

  new LearningStep(
    38,
    undefined,

    new Question(
      "What is the value of b?",
      [false,-10,10],
      ["","The answer is not a number, just a boolean value.","
            The answer is not a number, just a boolean value."]
            ) ),

  new LearningStep(39, undefined, undefined),

  new LearningStep(40, undefined, undefined),

  new LearningStep(41, undefined, undefined),

  new LearningStep(42, undefined, undefined),

  new LearningStep(43, undefined, undefined),

  new LearningStep(44, undefined, undefined),

  new LearningStep(45, undefined, undefined)]
                    ) /* end LearningSteps */
                    ) /* end Action */,
new Action (
                    new Program(`

var is_it_summer = true;
var is_it_winter = true;
var age_under_18 = true;
var before_8pm = true;
var after_7am = false;

var x = 10;
var x_is_greater_than_1 = true;
x = 0;
x_is_greater_than_1;
x_is_greater_than_1 = false;
x = 100;
x_is_greater_than_1;




          `
          ) /* end Program */,
                  new LearningSteps(
      /* Learning Steps */
      [
new LearningStep(
  0,
  "## What are booleans used for?\n\n
  The computer uses them\nto make *decisions* in\n**if**
        statements and **loops**.\nWe'll learn about those
        later.\n\n
  True means 'yes' and false means 'no'.\nBooleans are very
        important in programs.\n\n
  To get an intuition for why,\nimagine what it would be like
        \nif you could not communicate yes or no to other
        people (no nodding, no synonyms like sure or nah).",
  undefined ),

new LearningStep(
  0,
  "## Naming Boolean Variables\n\n
  Here's the style that\npeople choose names for\nboolean
        variables, just\nto give you an idea.",
  undefined ),

new LearningStep(0, undefined, undefined),

new LearningStep(1, undefined, undefined),

new LearningStep(
  1,
  "You might see a variable like this in a program that
        decides what background to show on a shopping website
        . We'll learn how to use a variable like this later."
        ,
  undefined ),

new LearningStep(2, undefined, undefined),

new LearningStep(3, undefined, undefined),

new LearningStep(4, undefined, undefined),

new LearningStep(5, undefined, undefined),

new LearningStep(6, undefined, undefined),

new LearningStep(7, undefined, undefined),

new LearningStep(8, undefined, undefined),

new LearningStep(9, undefined, undefined),

new LearningStep(
  9,
  undefined,

  new Question(
    "What value does is_it_winter have?",
    [false,10,-10],
    ["Look at the code to the right of the equal sign next to
          is_it_winter. What value does that put on the
          stack?","The answer is a boolean, not a number.","
          The answer is a boolean, not a number."] ) ),

new LearningStep(
  9,
  " ## No context\n\n
  Notice that the variable is_it_summer is true and the
        variable is_it_winter is also true. We humans know
        that **it can't be winter and summer at the same time
        .**\n\n
  The computer just follows rules to evaluate the expression
        to the right of the = and sets the variable to that
        value. **The name of the variable doesn't change how
        the computer treats it.**",
  undefined ),
```

```
new LearningStep(
  10,
  " ",
  undefined ),

new LearningStep(11, undefined, undefined),

new LearningStep(12, undefined, undefined),

new LearningStep(13, undefined, undefined),

new LearningStep(14, undefined, undefined),

new LearningStep(15, undefined, undefined),

new LearningStep(16, undefined, undefined),

new LearningStep(17, undefined, undefined),

new LearningStep(18, undefined, undefined),

new LearningStep(19, undefined, undefined),

new LearningStep(
  19,
  "What value does before_8pm have?",

  new Question(
    "What value does before_8pm have?",
    [false,-10,10],
    ["","The answer is a boolean, not a number.","The answer
        is a boolean, not a number."] ) ),

new LearningStep(20, undefined, undefined),

new LearningStep(21, undefined, undefined),

new LearningStep(22, undefined, undefined),

new LearningStep(23, undefined, undefined),

new LearningStep(23, undefined, undefined),

new LearningStep(24, undefined, undefined),

new LearningStep(25, undefined, undefined),

new LearningStep(26, undefined, undefined),

new LearningStep(27, undefined, undefined),

new LearningStep(28, undefined, undefined),

new LearningStep(29, undefined, undefined),

new LearningStep(30, undefined, undefined),

new LearningStep(31, undefined, undefined),

new LearningStep(32, undefined, undefined),

new LearningStep(33, undefined, undefined),

new LearningStep(34, undefined, undefined),

new LearningStep(35, undefined, undefined),

new LearningStep(36, undefined, undefined),

new LearningStep(37, undefined, undefined),

new LearningStep(38, undefined, undefined),

new LearningStep(
  39,
  " ",

  new Question(
    "What is the value of x_is_greater_than_1?",
    [false,10,-10],
    ["While you as a human can tell that x's value is less
        than 1, the computer doesn't change this variable's
         value unless told to do so.","The answer is a
        boolean, not a number.","The answer is a boolean,
        not a number."] ) ),

new LearningStep(39, undefined, undefined),

new LearningStep(40, undefined, undefined),

new LearningStep(41, undefined, undefined),

new LearningStep(42, undefined, undefined),

new LearningStep(43, undefined, undefined),

new LearningStep(44, undefined, undefined),

new LearningStep(45, undefined, undefined),

new LearningStep(46, undefined, undefined),

new LearningStep(
  47,
  undefined,

  new Question(
    " What value is on the stack?",
    [true,10,-10],
    ["While you as a human can tell that x's value is greater
        than 1, the computer doesn't change this variable'
        s value unless told to do so.","The answer is a
        boolean, not a number.","The answer is a boolean,
        not a number."] ) ),

new LearningStep(
  47,
  "Boolean variable names may have meaning to people, but the
       computer doesn't change their values unless code
      explicitly does so. It's the responsibility of
      programmers to write **understandable** code where
      the **names of variables relate to their values**. \n
      \n
  Variables with boolean values work like any other variables
       - if you use them in an experssion, they put a value
       on the stack.\n\n
  ",
  undefined ),

new LearningStep(
  48,
  "Boolean variable names may have meaning to people, but the
       computer doesn't change their values unless code
      explicitly does so. It's the responsibility of
      programmers to write **understandable** code where
      the **names of variables relate to their values**. \n
      \n
  Variables with boolean values work like any other variables
       - if you use them in an expression, they put a value
       on the stack.\n\n
  ",
  undefined ),

new LearningStep(
  48,
  "Onwards!",
  undefined )]
                ) /* end LearningSteps */
                ) /* end Action */,
new Action (
                new Program(`

var a = true;
var b = false;

a = b;
b = a;

a = false;
b = false;
var c = true;
var d = b;

a = c;
b = a;
c = d;
d = a;

var got_better_at_vars = true;
```

```
          `
          ) /* end Program */,
                  new LearningSteps(
      /* Learning Steps */
      [
new LearningStep(
  0,
  "## Changing Values\n\n
  Let's check that we understand using and assigning
       variables.\n\n
  ",
  undefined ),

new LearningStep(
  1,
  "",
  undefined ),

new LearningStep(2, undefined, undefined),

new LearningStep(3, undefined, undefined),

new LearningStep(4, undefined, undefined),

new LearningStep(5, undefined, undefined),

new LearningStep(6, undefined, undefined),

new LearningStep(7, undefined, undefined),

new LearningStep(8, undefined, undefined),

new LearningStep(9, undefined, undefined),

new LearningStep(10, undefined, undefined),

new LearningStep(11, undefined, undefined),

new LearningStep(12, undefined, undefined),

new LearningStep(13, undefined, undefined),

new LearningStep(14, undefined, undefined),

new LearningStep(15, undefined, undefined),

new LearningStep(16, undefined, undefined),

new LearningStep(
  16,
  undefined,

  new Question(
    "What is the value of b?",
    [true,-10,10],
    ["a's value was changed to b's value. The new value in a
          the got assigned to b, which happened to be the
          same value b originally had. Step backwards to
          watch that again after you complete this question."
          ,"The answer is a boolean, not a number.","The
          answer is a boolean, not a number."] ) ),

new LearningStep(17, undefined, undefined),

new LearningStep(18, undefined, undefined),

new LearningStep(19, undefined, undefined),

new LearningStep(20, undefined, undefined),

new LearningStep(21, undefined, undefined),

new LearningStep(22, undefined, undefined),

new LearningStep(23, undefined, undefined),

new LearningStep(24, undefined, undefined),

new LearningStep(25, undefined, undefined),

new LearningStep(26, undefined, undefined),

new LearningStep(27, undefined, undefined),

new LearningStep(28, undefined, undefined),

new LearningStep(29, undefined, undefined),

new LearningStep(30, undefined, undefined),

new LearningStep(31, undefined, undefined),

new LearningStep(32, undefined, undefined),

new LearningStep(33, undefined, undefined),

new LearningStep(34, undefined, undefined),

new LearningStep(35, undefined, undefined),

new LearningStep(36, undefined, undefined),

new LearningStep(37, undefined, undefined),

new LearningStep(38, undefined, undefined),

new LearningStep(39, undefined, undefined),

new LearningStep(39, undefined, undefined),

new LearningStep(40, undefined, undefined),

new LearningStep(41, undefined, undefined),

new LearningStep(42, undefined, undefined),

new LearningStep(43, undefined, undefined),

new LearningStep(44, undefined, undefined),

new LearningStep(
  44,
  undefined,

  new Question(
    "What is the value of d?",
    [false,-10,10],
    ["","The answer is a boolean, not a number.","The answer
          is a boolean, not a number."] ) ),

new LearningStep(
  44,
  undefined,

  new Question(
    "What is the value of c?",
    [true,10,-10],
    ["","The answer is a boolean, not a number.","The answer
          is a boolean, not a number."] ) ),

new LearningStep(45, undefined, undefined),

new LearningStep(46, undefined, undefined),

new LearningStep(47, undefined, undefined),

new LearningStep(
  48,
  "Great job! Forever forward!",
  undefined )]
                  ) /* end LearningSteps */
                  ) /* end Action */]),
new KnowledgeUnit("Arrays",
          "Arrays are like a box of boxes.  They can have
               many positions to put values into.Your write
               an array of true then false as [true, false].
                If the array is named x, x[0] is the first
               position, x[1] is the second. You can getthe
               number of positions in an array by x.length.
               This puts a number value on the stack.",
          [
new Action (
                  new Program(`


var free_first_day = true;
var free_second_day = false;
var free_third_day = true;
```

```
var free_fourth_day = true;
var free_fifth_day = false;
var free_sixth_day = true;
var free_seventh_day = false;



[ true, false];



[ true, false ];


[ false, true ];


[ true, true, false];



[ true ];


true;

[true];
true;
[false];
false;



var an_array = [ true, false ];
var another = [ false, true];
var third = [ true, true, false];



var an_array = [ true, false ];
var another = [ false, true];
an_array;
another;


var joe = [ true ];
joe;

var amy = [false];
amy;
joe;



var free_first_day = true;
var free_second_day = false;
var free_third_day = true;
var free_fourth_day = true;
var free_fifth_day = false;
var free_sixth_day = true;
var free_seventh_day = false;

var free_day_first = true;
var free_day_second = false;
var free_day_third = true;
var free_day_fourth = true;
var free_day_fifth = false;
var free_day_sixth = true;
var free_day_seventh = false;

var day_is_free_first = true;
var day_is_free_second = false;
var day_is_free_third = true;
var day_is_free_fourth = true;
var day_is_free_fifth = false;
var day_is_free_sixth = true;
var day_is_free_seventh = false;


var day_is_free =
    [ true, false, true, true,
      false, true, false ];
```

```
          ,
          ) /* end Program */,
                  new LearningSteps(
      /* Learning Steps */
      [
new LearningStep(
  0,
  "## Arrays\n\n
  Variables are like a box that\ncan hold a single value.\n\n
  If we want to store many values,\nwe need to make many
       variables.\n\n
  Let's say we want to remember\nwhat days you are available\
       nin a scheduling program for\nsome week. No one wants
        to\ntell someone they are free\nwhen they're
       actually busy\nand have to cancel, and the\ncomputer
       is great at\nremembering things!\n\n
  Let's say you're free on\nthe first, third, fourth,\nand
       sixth day of this week.\nYou might store this like:",
  undefined ),

new LearningStep(0, undefined, undefined),

new LearningStep(1, undefined, undefined),

new LearningStep(2, undefined, undefined),

new LearningStep(3, undefined, undefined),

new LearningStep(4, undefined, undefined),

new LearningStep(5, undefined, undefined),

new LearningStep(6, undefined, undefined),

new LearningStep(7, undefined, undefined),

new LearningStep(8, undefined, undefined),

new LearningStep(9, undefined, undefined),

new LearningStep(10, undefined, undefined),

new LearningStep(11, undefined, undefined),

new LearningStep(12, undefined, undefined),

new LearningStep(13, undefined, undefined),

new LearningStep(14, undefined, undefined),

new LearningStep(15, undefined, undefined),

new LearningStep(16, undefined, undefined),

new LearningStep(17, undefined, undefined),

new LearningStep(18, undefined, undefined),

new LearningStep(19, undefined, undefined),

new LearningStep(20, undefined, undefined),

new LearningStep(21, undefined, undefined),

new LearningStep(22, undefined, undefined),

new LearningStep(23, undefined, undefined),

new LearningStep(24, undefined, undefined),

new LearningStep(25, undefined, undefined),

new LearningStep(26, undefined, undefined),

new LearningStep(27, undefined, undefined),

new LearningStep(28, undefined, undefined),

new LearningStep(29, undefined, undefined),

new LearningStep(30, undefined, undefined),
```

```
new LearningStep(31, undefined, undefined),

new LearningStep(32, undefined, undefined),

new LearningStep(33, undefined, undefined),

new LearningStep(34, undefined, undefined),

new LearningStep(
  34,
  "This takes a lot of time to\ntype and is fairly long. For\
       na real calendar program,\nthe whole month\nwould
       take about 30 days, the \nwhole year 365 days. That
       would\nbe a lot of variables to name\nand type.\n\n
  **Arrays** help us do this better.\n\n
  **Arrays** are like a box of boxes.\nThey can hold many
       values;\neach value gets its own\nposition.\n\n
  For an array that holds two values,\nthere is a first place
        in the array,\nand a second place. Let's look at an
       array with true in the first position and false in
       the second position.",
  undefined ),

new LearningStep(35, undefined, undefined),

new LearningStep(36, undefined, undefined),

new LearningStep(37, undefined, undefined),

new LearningStep(38, undefined, undefined),

new LearningStep(
  39,
  " ",
  undefined ),

new LearningStep(
  40,
  "Arrays are written like this:\nthe [ indicates the
       beginning\nof an array, followed\nby the values to
       put in\nthe array. A comma (,)\nis used to tell when
       one\nplace ends. The ] indicates\nthat the array ends
       .",
  undefined ),

new LearningStep(40, undefined, undefined),

new LearningStep(41, undefined, undefined),

new LearningStep(42, undefined, undefined),

new LearningStep(43, undefined, undefined),

new LearningStep(44, undefined, undefined),

new LearningStep(
  45,
  "This line puts an array on the stack that holds false then
       true.",
  undefined ),

new LearningStep(45, undefined, undefined),

new LearningStep(46, undefined, undefined),

new LearningStep(47, undefined, undefined),

new LearningStep(48, undefined, undefined),

new LearningStep(
  48,
  undefined,

  new Question(
    "What value is on the stack after this instruction
         executes?",
    [[],false,[true,false]],
    ["This is an empty array. The actual array is not empty."
         ,"This is only the first value of the array.","This
          array has the correct values but they aren't in
         this order. The computer takes the top element of
         the stack and makes it the last element of the
         array, filling it up from back to front."] ) ),

new LearningStep(49, undefined, undefined),

new LearningStep(
  50,
  "this holds true then true then false",
  undefined ),

new LearningStep(50, undefined, undefined),

new LearningStep(51, undefined, undefined),

new LearningStep(52, undefined, undefined),

new LearningStep(53, undefined, undefined),

new LearningStep(54, undefined, undefined),

new LearningStep(55, undefined, undefined),

new LearningStep(
  55,
  undefined,

  new Question(
    "What value is on the stack?",
    [true,[false,true,true],[]],
    ["This is only the first value of the array.","This array
         has the correct values but they aren't in this
        order.","This is an empty array. The actual array
        is not empty."] ) ),

new LearningStep(
  55,
  "Arrays are just another\nkind of value. They go on and\
       noff the stack just like\nnumbers or booleans.\nThey
       can also be stored in\na variable.\n\n
  You can spot an array because\nit has [ ]",
  undefined ),

new LearningStep(
  56,
  "This is an array that\nonly has one position\nand it
       stores true",
  undefined ),

new LearningStep(56, undefined, undefined),

new LearningStep(57, undefined, undefined),

new LearningStep(58, undefined, undefined),

new LearningStep(59, undefined, undefined),

new LearningStep(
  60,
  "The value true is different from an  array value with one
       position that holds true.\nTry going back and forth\
       nand see the difference.",
  undefined ),

new LearningStep(60, undefined, undefined),

new LearningStep(61, undefined, undefined),

new LearningStep(62, undefined, undefined),

new LearningStep(63, undefined, undefined),

new LearningStep(64, undefined, undefined),

new LearningStep(65, undefined, undefined),

new LearningStep(
  65,
  undefined,

  new Question(
    "What value is on the stack?",
    [true,[true,true],[]],
    ["This was pushed as an array value. You selected the
         true value not inside an array.","This array has
         too many values in it.","This is an empty array.
         The actual array is not empty."] ) ),

new LearningStep(66, undefined, undefined),
```

```
new LearningStep(67, undefined, undefined),

new LearningStep(
  67,
  undefined,

  new Question(
    "What value is on the stack before this instruction
         executes?",
    [false,[true],[false]],
    ["","The answer is not an array, just a boolean value.","
         The answer is not an array, just a boolean value."]
     )),

new LearningStep(68, undefined, undefined),

new LearningStep(69, undefined, undefined),

new LearningStep(70, undefined, undefined),

new LearningStep(71, undefined, undefined),

new LearningStep(
  71,
  undefined,

  new Question(
    "What value is on the stack before this instruction
         executes?",
    [false,[],[false,false]],
    ["This was pushed as an array value. This is the false
         value not inside an array.","This is an empty array
         . The actual array is not empty.","This array has
         too many values in it."] )),

new LearningStep(72, undefined, undefined),

new LearningStep(73, undefined, undefined),

new LearningStep(
  74,
  "Let's store some arrays\ninside variables.",
  undefined ),

new LearningStep(74, undefined, undefined),

new LearningStep(75, undefined, undefined),

new LearningStep(76, undefined, undefined),

new LearningStep(77, undefined, undefined),

new LearningStep(78, undefined, undefined),

new LearningStep(79, undefined, undefined),

new LearningStep(80, undefined, undefined),

new LearningStep(81, undefined, undefined),

new LearningStep(82, undefined, undefined),

new LearningStep(83, undefined, undefined),

new LearningStep(84, undefined, undefined),

new LearningStep(85, undefined, undefined),

new LearningStep(86, undefined, undefined),

new LearningStep(87, undefined, undefined),

new LearningStep(88, undefined, undefined),

new LearningStep(89, undefined, undefined),

new LearningStep(90, undefined, undefined),

new LearningStep(91, undefined, undefined),

new LearningStep(92, undefined, undefined),

new LearningStep(93, undefined, undefined),

new LearningStep(94, undefined, undefined),

new LearningStep(95, undefined, undefined),

new LearningStep(96, undefined, undefined),

new LearningStep(97, undefined, undefined),

new LearningStep(98, undefined, undefined),

new LearningStep(
  98,
  undefined,

  new Question(
    "What value does the variable third have?",
    [[],[false,true,true],true],
    ["This is an empty array. The actual array is not empty."
         ,"This array has the correct values but they aren't
          in this order.","This is only the first value of
         the array."] ) ),

new LearningStep(99, undefined, undefined),

new LearningStep(100, undefined, undefined),

new LearningStep(101, undefined, undefined),

new LearningStep(102, undefined, undefined),

new LearningStep(103, undefined, undefined),

new LearningStep(104, undefined, undefined),

new LearningStep(105, undefined, undefined),

new LearningStep(106, undefined, undefined),

new LearningStep(107, undefined, undefined),

new LearningStep(108, undefined, undefined),

new LearningStep(109, undefined, undefined),

new LearningStep(110, undefined, undefined),

new LearningStep(111, undefined, undefined),

new LearningStep(112, undefined, undefined),

new LearningStep(113, undefined, undefined),

new LearningStep(114, undefined, undefined),

new LearningStep(
  114,
  "Next, we'll use a variable to put an array on the stack.\
       nYou can tell that because\nyou see the [ ] in\n[true
       , false].",
  undefined ),

new LearningStep(115, undefined, undefined),

new LearningStep(116, undefined, undefined),

new LearningStep(
  116,
  undefined,

  new Question(
    "What value is on the stack before this instruction
         executes?",
    [true,[],[false,true]],
    ["This is only the first value of the array.","This is an
          empty array. The actual array is not empty.","This
          array has the correct values but they aren't in
         this order."] ) ),

new LearningStep(117, undefined, undefined),

new LearningStep(118, undefined, undefined),

new LearningStep(
  118,
  undefined,
```

```
  new Question(
    "What value is on the stack before this instruction
         executes?",
    [[true,false],[],false],
    ["This array has the correct values but they aren't in
         this order.","This is an empty array. The actual
         array is not empty.","This is only the first value
         of the array."] ) ),

new LearningStep(
  119,
  "This is how to put\nthe array value on the stack.\nYou can
       tell that because\nyou see [true]",
  undefined ),

new LearningStep(119, undefined, undefined),

new LearningStep(120, undefined, undefined),

new LearningStep(121, undefined, undefined),

new LearningStep(122, undefined, undefined),

new LearningStep(123, undefined, undefined),

new LearningStep(124, undefined, undefined),

new LearningStep(125, undefined, undefined),

new LearningStep(126, undefined, undefined),

new LearningStep(127, undefined, undefined),

new LearningStep(128, undefined, undefined),

new LearningStep(129, undefined, undefined),

new LearningStep(130, undefined, undefined),

new LearningStep(131, undefined, undefined),

new LearningStep(132, undefined, undefined),

new LearningStep(133, undefined, undefined),

new LearningStep(134, undefined, undefined),

new LearningStep(135, undefined, undefined),

new LearningStep(
  135,
  undefined,

  new Question(
    "What value is on the stack after this instruction
         executes?",
    [[false,false],[],10],
    ["This array has too many values in it.","This is an
         empty array. The actual array is not empty.","This
         is not an array."] ) ),

new LearningStep(136, undefined, undefined),

new LearningStep(
  136,
  undefined,

  new Question(
    "What is the value of the joe variable right now?",
    [true,10,[true,true]],
    ["The variable joe is holding an array; this is the value
          in the array but joe's value is an array.","This
         is not an array.","This array has too many values
         in it."] ) ),

new LearningStep(137, undefined, undefined),

new LearningStep(138, undefined, undefined),

new LearningStep(
  139,
  "## Why use arrays\n\n
  Let's compare how we might store the week using 7 variables
       and how we can with an array.\n\n
  To motivate this, let's start with the day variables and
       write them with different names a few times.",
  undefined ),

new LearningStep(139, undefined, undefined),

new LearningStep(140, undefined, undefined),

new LearningStep(141, undefined, undefined),

new LearningStep(142, undefined, undefined),

new LearningStep(143, undefined, undefined),

new LearningStep(144, undefined, undefined),

new LearningStep(145, undefined, undefined),

new LearningStep(146, undefined, undefined),

new LearningStep(147, undefined, undefined),

new LearningStep(148, undefined, undefined),

new LearningStep(149, undefined, undefined),

new LearningStep(150, undefined, undefined),

new LearningStep(151, undefined, undefined),

new LearningStep(152, undefined, undefined),

new LearningStep(153, undefined, undefined),

new LearningStep(154, undefined, undefined),

new LearningStep(155, undefined, undefined),

new LearningStep(156, undefined, undefined),

new LearningStep(157, undefined, undefined),

new LearningStep(158, undefined, undefined),

new LearningStep(159, undefined, undefined),

new LearningStep(160, undefined, undefined),

new LearningStep(161, undefined, undefined),

new LearningStep(162, undefined, undefined),

new LearningStep(163, undefined, undefined),

new LearningStep(164, undefined, undefined),

new LearningStep(165, undefined, undefined),

new LearningStep(166, undefined, undefined),

new LearningStep(167, undefined, undefined),

new LearningStep(168, undefined, undefined),

new LearningStep(169, undefined, undefined),

new LearningStep(170, undefined, undefined),

new LearningStep(171, undefined, undefined),

new LearningStep(172, undefined, undefined),

new LearningStep(173, undefined, undefined),

new LearningStep(
  174,
  "Now we will move the first, second, etc to the end of the
       name",
  undefined ),

new LearningStep(174, undefined, undefined),

new LearningStep(175, undefined, undefined),

new LearningStep(176, undefined, undefined),
```

```
new LearningStep(177, undefined, undefined),
new LearningStep(178, undefined, undefined),
new LearningStep(179, undefined, undefined),
new LearningStep(180, undefined, undefined),
new LearningStep(181, undefined, undefined),
new LearningStep(182, undefined, undefined),
new LearningStep(183, undefined, undefined),
new LearningStep(184, undefined, undefined),
new LearningStep(185, undefined, undefined),
new LearningStep(186, undefined, undefined),
new LearningStep(187, undefined, undefined),
new LearningStep(188, undefined, undefined),
new LearningStep(189, undefined, undefined),
new LearningStep(190, undefined, undefined),
new LearningStep(191, undefined, undefined),
new LearningStep(192, undefined, undefined),
new LearningStep(193, undefined, undefined),
new LearningStep(194, undefined, undefined),
new LearningStep(195, undefined, undefined),
new LearningStep(196, undefined, undefined),
new LearningStep(197, undefined, undefined),
new LearningStep(198, undefined, undefined),
new LearningStep(199, undefined, undefined),
new LearningStep(200, undefined, undefined),
new LearningStep(201, undefined, undefined),
new LearningStep(202, undefined, undefined),
new LearningStep(203, undefined, undefined),
new LearningStep(204, undefined, undefined),
new LearningStep(205, undefined, undefined),
new LearningStep(206, undefined, undefined),
new LearningStep(207, undefined, undefined),
new LearningStep(208, undefined, undefined),
new LearningStep(
  209,
  "We will to name the array version of the week as *
       day_is_free*. Let's make the names appear similar so
       we can see the pattern clearly.",
  undefined ),
new LearningStep(209, undefined, undefined),
new LearningStep(210, undefined, undefined),
new LearningStep(211, undefined, undefined),
new LearningStep(212, undefined, undefined),
new LearningStep(213, undefined, undefined),
new LearningStep(214, undefined, undefined),
new LearningStep(215, undefined, undefined),
new LearningStep(216, undefined, undefined),
new LearningStep(217, undefined, undefined),
new LearningStep(218, undefined, undefined),
new LearningStep(219, undefined, undefined),
new LearningStep(220, undefined, undefined),
new LearningStep(221, undefined, undefined),
new LearningStep(222, undefined, undefined),
new LearningStep(223, undefined, undefined),
new LearningStep(224, undefined, undefined),
new LearningStep(225, undefined, undefined),
new LearningStep(226, undefined, undefined),
new LearningStep(227, undefined, undefined),
new LearningStep(228, undefined, undefined),
new LearningStep(229, undefined, undefined),
new LearningStep(230, undefined, undefined),
new LearningStep(231, undefined, undefined),
new LearningStep(232, undefined, undefined),
new LearningStep(233, undefined, undefined),
new LearningStep(234, undefined, undefined),
new LearningStep(235, undefined, undefined),
new LearningStep(236, undefined, undefined),
new LearningStep(237, undefined, undefined),
new LearningStep(238, undefined, undefined),
new LearningStep(239, undefined, undefined),
new LearningStep(240, undefined, undefined),
new LearningStep(241, undefined, undefined),
new LearningStep(242, undefined, undefined),
new LearningStep(243, undefined, undefined),
new LearningStep(
  244,
  "",
  undefined ),
new LearningStep(
  245,
  "",
  undefined ),
new LearningStep(
  246,
  "",
  undefined ),
new LearningStep(
  247,
  "",
  undefined ),
new LearningStep(
  248,
  "",
  undefined ),
new LearningStep(
  249,
  "",
  undefined ),
```

```
  new LearningStep(
    250,
    "",
    undefined ),

  new LearningStep(
    251,
    "",
    undefined ),

  new LearningStep(
    252,
    "",
    undefined ),

  new LearningStep(
    253,
    "",
    undefined ),

  new LearningStep(
    254,
    "",
    undefined ),

  new LearningStep(
    255,
    "",
    undefined ),

  new LearningStep(
    257,
    "",
    undefined ),

  new LearningStep(
    256,
    "",
    undefined ),

  new LearningStep(
    256,
    "## Array Advantages\n\n
    With an array it is much shorter!\nYou have to type 189
          letters\nfor separate variables, compared\nto 72 for
          an array.\n\n
    This is one reason the language designers added arrays to
          the language - it lets people write shorter programs
          with less repeating of words. Shorter programs (as
          long as they are still understandable) take people **
          less time to read and write**.\n\n
    Language designers want to save people's time, but you can
          see that **the cost is you have to teach people how
          arrays work**. I think it is worth it and you'll see
          later more reasons why arrays are useful.\n\n
    Next, we will learn how to access\nthe values inside the
          array\nand put them on the stack.\n
                                ",
    undefined )]
                  ) /* end LearningSteps */
                  ) /* end Action */,
new Action (
                  new Program(`

var day_is_free_first = true;
var day_is_free_second = false;
var day_is_free_third = true;
var day_is_free_fourth = true;
var day_is_free_fifth = false;
var day_is_free_sixth = true;
var day_is_free_seventh = false;


var day_is_free =
    [ true, false, true, true,
      false, true, false ];




var day_is_free_1 = true;
var day_is_free_2 = false;
var day_is_free_3 = true;
var day_is_free_4 = true;
var day_is_free_5 = false;
var day_is_free_6 = true;
var day_is_free_7 = false;



var day_is_free_0 = true;
var day_is_free_1 = false;
var day_is_free_2 = true;
var day_is_free_3 = true;
var day_is_free_4 = false;
var day_is_free_5 = true;
var day_is_free_6 = false;



var day_is_free_0 = true;
var day_is_free_1 = false;
var day_is_free_2 = true;
var day_is_free_3 = true;
var day_is_free_4 = false;
var day_is_free_5 = true;
var day_is_free_6 = false;

var day_is_free =
    [ true, false, true, true,
      false, true, false ];



day_is_free_0;
day_is_free[0];


day_is_free_0 = false;


day_is_free_0;


day_is_free[0];

day_is_free_0;
day_is_free[0];


var day_is_free_0 = true;
var day_is_free_1 = false;
var day_is_free_2 = true;
var day_is_free_3 = true;
var day_is_free_4 = false;
var day_is_free_5 = true;
var day_is_free_6 = false;

var day_is_free =
    [ true, false, true, true,
      false, true, false ];

day_is_free_0;
day_is_free[0];


day_is_free_1;
day_is_free[1];


day_is_free_2;
day_is_free[2];


day_is_free_3;
day_is_free[3];


day_is_free_4;
day_is_free[4];


day_is_free_5;
day_is_free[5];


day_is_free_6;
day_is_free[6];
```

```
var c = [true, false];

c[0];

c[1];

var d = [true, false];
d[0];
d[1];

var e = [false, true, true];
e[0];
e[1];
e[2];
```

```
            `
            ) /* end Program */,
                    new LearningSteps(
        /* Learning Steps */
        [
  new LearningStep(
    0,
    "## Looking Up Array Values\n\n
    To motivate this, let's start with the day variables and
         write them with different names a few times. We
         repeated the last lesson in case you want to look it
         over again.\n\n
    It's a way to represent\nthat you're free on the first,
         third, fourth, and sixth day of this week.",
    undefined ),

  new LearningStep(0, undefined, undefined),

  new LearningStep(1, undefined, undefined),

  new LearningStep(2, undefined, undefined),

  new LearningStep(3, undefined, undefined),

  new LearningStep(4, undefined, undefined),

  new LearningStep(5, undefined, undefined),

  new LearningStep(6, undefined, undefined),

  new LearningStep(7, undefined, undefined),

  new LearningStep(8, undefined, undefined),

  new LearningStep(9, undefined, undefined),

  new LearningStep(10, undefined, undefined),

  new LearningStep(11, undefined, undefined),

  new LearningStep(12, undefined, undefined),

  new LearningStep(13, undefined, undefined),

  new LearningStep(14, undefined, undefined),

  new LearningStep(15, undefined, undefined),

  new LearningStep(16, undefined, undefined),

  new LearningStep(17, undefined, undefined),

  new LearningStep(18, undefined, undefined),

  new LearningStep(19, undefined, undefined),

  new LearningStep(20, undefined, undefined),

  new LearningStep(21, undefined, undefined),

  new LearningStep(22, undefined, undefined),

  new LearningStep(23, undefined, undefined),

  new LearningStep(24, undefined, undefined),

  new LearningStep(25, undefined, undefined),

  new LearningStep(26, undefined, undefined),

  new LearningStep(27, undefined, undefined),

  new LearningStep(28, undefined, undefined),

  new LearningStep(29, undefined, undefined),

  new LearningStep(30, undefined, undefined),

  new LearningStep(31, undefined, undefined),

  new LearningStep(32, undefined, undefined),

  new LearningStep(33, undefined, undefined),

  new LearningStep(34, undefined, undefined),

  new LearningStep(
    35,
    "Here's another way to store that using an array",
    undefined ),

  new LearningStep(35, undefined, undefined),

  new LearningStep(36, undefined, undefined),

  new LearningStep(37, undefined, undefined),

  new LearningStep(38, undefined, undefined),

  new LearningStep(39, undefined, undefined),

  new LearningStep(40, undefined, undefined),

  new LearningStep(41, undefined, undefined),

  new LearningStep(42, undefined, undefined),

  new LearningStep(43, undefined, undefined),

  new LearningStep(44, undefined, undefined),

  new LearningStep(45, undefined, undefined),

  new LearningStep(46, undefined, undefined),

  new LearningStep(47, undefined, undefined),

  new LearningStep(
    48,
    "Soon, we will learn how to access the values inside the
         array and put them on the stack!\n\n
    Now let's rewrite the day variables again, look for the
         pattern",
    undefined ),

  new LearningStep(48, undefined, undefined),

  new LearningStep(49, undefined, undefined),

  new LearningStep(50, undefined, undefined),

  new LearningStep(51, undefined, undefined),

  new LearningStep(52, undefined, undefined),

  new LearningStep(53, undefined, undefined),

  new LearningStep(54, undefined, undefined),

  new LearningStep(55, undefined, undefined),

  new LearningStep(56, undefined, undefined),

  new LearningStep(57, undefined, undefined),

  new LearningStep(58, undefined, undefined),

  new LearningStep(59, undefined, undefined),
```

```
new LearningStep(60, undefined, undefined),
new LearningStep(61, undefined, undefined),
new LearningStep(62, undefined, undefined),
new LearningStep(63, undefined, undefined),
new LearningStep(64, undefined, undefined),
new LearningStep(65, undefined, undefined),
new LearningStep(66, undefined, undefined),
new LearningStep(67, undefined, undefined),
new LearningStep(68, undefined, undefined),
new LearningStep(69, undefined, undefined),
new LearningStep(70, undefined, undefined),
new LearningStep(71, undefined, undefined),
new LearningStep(72, undefined, undefined),
new LearningStep(73, undefined, undefined),
new LearningStep(74, undefined, undefined),
new LearningStep(75, undefined, undefined),
new LearningStep(76, undefined, undefined),
new LearningStep(77, undefined, undefined),
new LearningStep(78, undefined, undefined),
new LearningStep(79, undefined, undefined),
new LearningStep(80, undefined, undefined),
new LearningStep(81, undefined, undefined),
new LearningStep(82, undefined, undefined),

new LearningStep(
  83,
  "That way of writing them still looks pretty similar. Let's
       rewrite them again.\nWe'll start with 0 instead of
      1.",
  undefined ),

new LearningStep(83, undefined, undefined),
new LearningStep(84, undefined, undefined),
new LearningStep(85, undefined, undefined),
new LearningStep(86, undefined, undefined),
new LearningStep(87, undefined, undefined),
new LearningStep(88, undefined, undefined),
new LearningStep(89, undefined, undefined),
new LearningStep(90, undefined, undefined),
new LearningStep(91, undefined, undefined),
new LearningStep(92, undefined, undefined),
new LearningStep(93, undefined, undefined),
new LearningStep(94, undefined, undefined),
new LearningStep(95, undefined, undefined),
new LearningStep(96, undefined, undefined),
new LearningStep(97, undefined, undefined),
new LearningStep(98, undefined, undefined),
new LearningStep(99, undefined, undefined),
new LearningStep(100, undefined, undefined),
new LearningStep(101, undefined, undefined),
new LearningStep(102, undefined, undefined),
new LearningStep(103, undefined, undefined),
new LearningStep(104, undefined, undefined),
new LearningStep(105, undefined, undefined),
new LearningStep(106, undefined, undefined),
new LearningStep(107, undefined, undefined),
new LearningStep(108, undefined, undefined),
new LearningStep(109, undefined, undefined),
new LearningStep(110, undefined, undefined),
new LearningStep(111, undefined, undefined),
new LearningStep(112, undefined, undefined),
new LearningStep(113, undefined, undefined),
new LearningStep(114, undefined, undefined),
new LearningStep(115, undefined, undefined),
new LearningStep(116, undefined, undefined),
new LearningStep(117, undefined, undefined),

new LearningStep(
  118,
  "This is still the seven days of the week. It's another way
       of representing that you're free on the first, third
      , fourth, and sixth day of this week.\n\n
  We just decided **to number the\nfirst day with a 0 instead
       of\nusing a 1.**\n\n
  ",
  undefined ),

new LearningStep(
  118,
  "Let's start with\nhow to get the value in the first\nplace
       in the array.\n\n
  We'll repeat the days variables\nand the array, so you can
      more easily see the program text as you continue down
      . Then we'll access the first day with a variable,
      and then with the array.\n\n
  Remember in the computer these variables don't have any
      relationship, we're just showing you a parallel
      example to help you understand looking up array
      values.",
  undefined ),

new LearningStep(118, undefined, undefined),
new LearningStep(119, undefined, undefined),
new LearningStep(120, undefined, undefined),
new LearningStep(121, undefined, undefined),
new LearningStep(122, undefined, undefined),
new LearningStep(123, undefined, undefined),
new LearningStep(124, undefined, undefined),
new LearningStep(125, undefined, undefined),
new LearningStep(126, undefined, undefined),
new LearningStep(127, undefined, undefined),
new LearningStep(128, undefined, undefined),
new LearningStep(129, undefined, undefined),
```

```
new LearningStep(130, undefined, undefined),

new LearningStep(131, undefined, undefined),

new LearningStep(132, undefined, undefined),

new LearningStep(133, undefined, undefined),

new LearningStep(134, undefined, undefined),

new LearningStep(135, undefined, undefined),

new LearningStep(136, undefined, undefined),

new LearningStep(137, undefined, undefined),

new LearningStep(138, undefined, undefined),

new LearningStep(139, undefined, undefined),

new LearningStep(140, undefined, undefined),

new LearningStep(141, undefined, undefined),

new LearningStep(142, undefined, undefined),

new LearningStep(143, undefined, undefined),

new LearningStep(144, undefined, undefined),

new LearningStep(145, undefined, undefined),

new LearningStep(146, undefined, undefined),

new LearningStep(147, undefined, undefined),

new LearningStep(148, undefined, undefined),

new LearningStep(149, undefined, undefined),

new LearningStep(150, undefined, undefined),

new LearningStep(151, undefined, undefined),

new LearningStep(152, undefined, undefined),

new LearningStep(153, undefined, undefined),

new LearningStep(154, undefined, undefined),

new LearningStep(155, undefined, undefined),

new LearningStep(156, undefined, undefined),

new LearningStep(157, undefined, undefined),

new LearningStep(158, undefined, undefined),

new LearningStep(159, undefined, undefined),

new LearningStep(160, undefined, undefined),

new LearningStep(161, undefined, undefined),

new LearningStep(162, undefined, undefined),

new LearningStep(163, undefined, undefined),

new LearningStep(164, undefined, undefined),

new LearningStep(165, undefined, undefined),

new LearningStep(
  166,
  "Here's getting the value for the first day using a
       variable.",
  undefined ),

new LearningStep(166, undefined, undefined),

new LearningStep(167, undefined, undefined),

new LearningStep(
  167,
  "Now watch the line \n\n
  day_is_free[0];\n\n
  and read what the **instructions** are doing and notice how
       **state** changes.\n\n
  You can read \"day_is_free[0]\" as\n\"the value at position
       0\nin day_is_free\"\nor\n\"the first value in the
       array\nstored in day_is_free\".\n\n
  We also call the position\nthe **index**.",
  undefined ),

new LearningStep(168, undefined, undefined),

new LearningStep(169, undefined, undefined),

new LearningStep(170, undefined, undefined),

new LearningStep(171, undefined, undefined),

new LearningStep(
  171,
  "Again, even though these names look very similar, the\
       ncomputer just sees them as\ninstructions that put
       values\non and off the stack. We'll\nshow this to you
       by changing\nthe value of one; the other's\nvalue **
       does not change**.",
  undefined ),

new LearningStep(172, undefined, undefined),

new LearningStep(173, undefined, undefined),

new LearningStep(174, undefined, undefined),

new LearningStep(
  175,
  "This line will look up a variable.",
  undefined ),

new LearningStep(175, undefined, undefined),

new LearningStep(176, undefined, undefined),

new LearningStep(
  177,
  "This line will look up variable day_is_free,\npush that
       array value onto the stack,\nand [0] makes an array
       lookup instruction.\n\n
  As we step through this line the computer will execute
       those instructions.",
  undefined ),

new LearningStep(177, undefined, undefined),

new LearningStep(178, undefined, undefined),

new LearningStep(179, undefined, undefined),

new LearningStep(180, undefined, undefined),

new LearningStep(
  180,
  "Look at the value on the stack. Then look at the first
       value in day_is_free in the namespace. They are the
       same, since that is the value that was looked up and
       put onto the stack.",
  undefined ),

new LearningStep(181, undefined, undefined),

new LearningStep(182, undefined, undefined),

new LearningStep(183, undefined, undefined),

new LearningStep(184, undefined, undefined),

new LearningStep(185, undefined, undefined),

new LearningStep(186, undefined, undefined),

new LearningStep(
  187,
  "We'll repeat declaring the day_is_free variables again, so
       you don't have to scroll far up if you want to look
       at them.\n\n
  Now we'll see how to access the other positions in arrays."
```

```
        ,
    undefined ),

new LearningStep(
    188,
    " ",
    undefined ),

new LearningStep(
    189,
    " ",
    undefined ),

new LearningStep(
    190,
    " ",
    undefined ),

new LearningStep(
    191,
    " ",
    undefined ),

new LearningStep(
    192,
    " ",
    undefined ),

new LearningStep(
    193,
    " ",
    undefined ),

new LearningStep(
    194,
    " ",
    undefined ),

new LearningStep(
    195,
    " ",
    undefined ),

new LearningStep(
    196,
    " ",
    undefined ),

new LearningStep(
    197,
    " ",
    undefined ),

new LearningStep(
    198,
    " ",
    undefined ),

new LearningStep(
    199,
    " ",
    undefined ),

new LearningStep(
    200,
    " ",
    undefined ),

new LearningStep(
    201,
    " ",
    undefined ),

new LearningStep(
    202,
    " ",
    undefined ),

new LearningStep(
    203,
    " ",
    undefined ),

new LearningStep(
    204,
    " ",
    undefined ),

new LearningStep(
    205,
    " ",
    undefined ),

new LearningStep(
    206,
    " ",
    undefined ),

new LearningStep(
    207,
    " ",
    undefined ),

new LearningStep(
    208,
    " ",
    undefined ),

new LearningStep(
    209,
    " ",
    undefined ),

new LearningStep(
    210,
    " ",
    undefined ),

new LearningStep(
    211,
    " ",
    undefined ),

new LearningStep(
    212,
    " ",
    undefined ),

new LearningStep(
    213,
    " ",
    undefined ),

new LearningStep(
    214,
    " ",
    undefined ),

new LearningStep(
    215,
    " ",
    undefined ),

new LearningStep(
    216,
    " ",
    undefined ),

new LearningStep(
    217,
    " ",
    undefined ),

new LearningStep(
    218,
    " ",
    undefined ),

new LearningStep(
    219,
    " ",
    undefined ),

new LearningStep(
    220,
    " ",
    undefined ),

new LearningStep(
    221,
```

```
    " ",
    undefined ),

new LearningStep(
    222,
    " ",
    undefined ),

new LearningStep(
    223,
    " ",
    undefined ),

new LearningStep(
    224,
    " ",
    undefined ),

new LearningStep(
    225,
    " ",
    undefined ),

new LearningStep(
    226,
    " ",
    undefined ),

new LearningStep(
    227,
    " ",
    undefined ),

new LearningStep(
    228,
    " ",
    undefined ),

new LearningStep(
    229,
    " ",
    undefined ),

new LearningStep(
    230,
    " ",
    undefined ),

new LearningStep(
    231,
    " ",
    undefined ),

new LearningStep(
    232,
    " ",
    undefined ),

new LearningStep(
    233,
    " ",
    undefined ),

new LearningStep(
    234,
    " ",
    undefined ),

new LearningStep(235, undefined, undefined),

new LearningStep(236, undefined, undefined),

new LearningStep(237, undefined, undefined),

new LearningStep(238, undefined, undefined),

new LearningStep(239, undefined, undefined),

new LearningStep(240, undefined, undefined),

new LearningStep(
    240,
    undefined,

    new Question(
        "What value is on the stack?",
        [false,[true],[false]],
        ["This gets the first value out of the array.","The
            answer is not an array, just a boolean value. The
            [0] takes it out of the array.","The answer is not
            an array, just a boolean value. The [0] takes it
            out. Look at the array definition to find the first
             value."] ) ),

new LearningStep(
    241,
    "This represents the second day",
    undefined ),

new LearningStep(241, undefined, undefined),

new LearningStep(242, undefined, undefined),

new LearningStep(
    242,
    "Now we'll see how to access the second position in the
        array.",
    undefined ),

new LearningStep(243, undefined, undefined),

new LearningStep(244, undefined, undefined),

new LearningStep(245, undefined, undefined),

new LearningStep(246, undefined, undefined),

new LearningStep(
    246,
    "Since 0 refers to the first position in the array, 1
        refers to the second position. ",
    undefined ),

new LearningStep(
    247,
    "This represents the third day",
    undefined ),

new LearningStep(247, undefined, undefined),

new LearningStep(248, undefined, undefined),

new LearningStep(249, undefined, undefined),

new LearningStep(250, undefined, undefined),

new LearningStep(251, undefined, undefined),

new LearningStep(252, undefined, undefined),

new LearningStep(
    253,
    "this is the fourth day",
    undefined ),

new LearningStep(253, undefined, undefined),

new LearningStep(254, undefined, undefined),

new LearningStep(255, undefined, undefined),

new LearningStep(256, undefined, undefined),

new LearningStep(257, undefined, undefined),

new LearningStep(
    257,
    undefined,

    new Question(
        "What value is on the stack after?",
        [false,[true],[false]],
        ["Look at the array again. Which position is the fourth (
            arrays start from 0)?","The answer is not an array,
             just a boolean value. The [3]  took the value out.
             ","The answer is not an array, just a boolean
            value. The [3] took the value out."] ) ),

new LearningStep(258, undefined, undefined),
```

```
new LearningStep(
  259,
  "this is the fifth day",
  undefined ),

new LearningStep(259, undefined, undefined),

new LearningStep(260, undefined, undefined),

new LearningStep(261, undefined, undefined),

new LearningStep(262, undefined, undefined),

new LearningStep(263, undefined, undefined),

new LearningStep(
  263,
  undefined,

  new Question(
    "What value is on the stack after?",
    [true,[true],[false]],
    ["Look at the array again. Which position is the fifth (
          arrays start from 0)?","The answer is not an array,
           just a boolean value. The [4]  took the value out.
           ","The answer is not an array, just a boolean
          value. The [4] took the value out."] ) ),

new LearningStep(264, undefined, undefined),

new LearningStep(
  265,
  "This is the sixth day",
  undefined ),

new LearningStep(265, undefined, undefined),

new LearningStep(266, undefined, undefined),

new LearningStep(267, undefined, undefined),

new LearningStep(268, undefined, undefined),

new LearningStep(269, undefined, undefined),

new LearningStep(270, undefined, undefined),

new LearningStep(
  271,
  "This is the seventh and final day.",
  undefined ),

new LearningStep(271, undefined, undefined),

new LearningStep(272, undefined, undefined),

new LearningStep(273, undefined, undefined),

new LearningStep(274, undefined, undefined),

new LearningStep(275, undefined, undefined),

new LearningStep(
  275,
  undefined,

  new Question(
    "What value is on the stack after?",
    [true,[false],[true]],
    ["[6] refers to the 7th value in the array (the last
          value). [0] is the first.","The answer is not an
          array, just a boolean value. The [6] after the
          variable takes the value out.","The answer is not
          an array, just a boolean value. The [6] after the
          variable takes the value out."] ) ),

new LearningStep(276, undefined, undefined),

new LearningStep(
  277,
  "Watch c[0];\nYou can read \"c[0]\" as\n\"the value at
       position 0 in c\"\nor\n\"the first value in the array
        in c\".",
  undefined ),

new LearningStep(277, undefined, undefined),

new LearningStep(278, undefined, undefined),

new LearningStep(279, undefined, undefined),

new LearningStep(280, undefined, undefined),

new LearningStep(281, undefined, undefined),

new LearningStep(282, undefined, undefined),

new LearningStep(283, undefined, undefined),

new LearningStep(284, undefined, undefined),

new LearningStep(285, undefined, undefined),

new LearningStep(286, undefined, undefined),

new LearningStep(287, undefined, undefined),

new LearningStep(288, undefined, undefined),

new LearningStep(
  288,
  undefined,

  new Question(
    "What is the value of c[0]?",
    [false,[false],[true]],
    ["c[0] refers to the first value in the array.","The
          answer is not an array, just a boolean value. The
          [0] after the variable takes the value out.","The
          answer is not an array, just a boolean value. The
          [0] after the variable takes the value out."] ) ),

new LearningStep(289, undefined, undefined),

new LearningStep(290, undefined, undefined),

new LearningStep(291, undefined, undefined),

new LearningStep(292, undefined, undefined),

new LearningStep(
  292,
  undefined,

  new Question(
    "What is the value of c[1]?",
    [true,[false],[true]],
    ["c[1] refers to the second (for this array, the last)
          value in the array.","The answer is not an array,
          just a boolean value. The [1] after the variable
          takes the value out.","The answer is not an array,
          just a boolean value. The [1] after the variable
          takes the value out."] ) ),

new LearningStep(293, undefined, undefined),

new LearningStep(294, undefined, undefined),

new LearningStep(295, undefined, undefined),

new LearningStep(296, undefined, undefined),

new LearningStep(297, undefined, undefined),

new LearningStep(298, undefined, undefined),

new LearningStep(299, undefined, undefined),

new LearningStep(300, undefined, undefined),

new LearningStep(301, undefined, undefined),

new LearningStep(302, undefined, undefined),

new LearningStep(303, undefined, undefined),

new LearningStep(304, undefined, undefined),

new LearningStep(
  304,
```

```
  undefined,

  new Question(
    "What value is on the stack before this instruction
         executes?",
    [false,[true],[false]],
    ["d[0] refers to the first value in the array.","The
         answer is not an array, just a boolean value. The
         [0] after the variable takes the value out.","The
         answer is not an array, just a boolean value. The
         [0] after the variable takes the value out."] ) ),

new LearningStep(305, undefined, undefined),

new LearningStep(306, undefined, undefined),

new LearningStep(307, undefined, undefined),

new LearningStep(308, undefined, undefined),

new LearningStep(
  308,
  undefined,

  new Question(
    "What value is on the stack before this instruction
         executes?",
    [true,[false],[true]],
    ["d[1] refers to the second (for this array, the last)
         value in the array.","The answer is not an array,
         just a boolean value. The [1] after the variable
         takes the value out.","The answer is not an array,
         just a boolean value. The [1] after the variable
         takes the value out."] ) ),

new LearningStep(309, undefined, undefined),

new LearningStep(310, undefined, undefined),

new LearningStep(311, undefined, undefined),

new LearningStep(312, undefined, undefined),

new LearningStep(313, undefined, undefined),

new LearningStep(314, undefined, undefined),

new LearningStep(315, undefined, undefined),

new LearningStep(316, undefined, undefined),

new LearningStep(317, undefined, undefined),

new LearningStep(318, undefined, undefined),

new LearningStep(319, undefined, undefined),

new LearningStep(320, undefined, undefined),

new LearningStep(321, undefined, undefined),

new LearningStep(322, undefined, undefined),

new LearningStep(323, undefined, undefined),

new LearningStep(324, undefined, undefined),

new LearningStep(325, undefined, undefined),

new LearningStep(326, undefined, undefined),

new LearningStep(327, undefined, undefined),

new LearningStep(328, undefined, undefined),

new LearningStep(
  328,
  undefined,

  new Question(
    "What value is on the stack after this instruction
         executes?",
    [false,[false],[true]],
    ["e[2] refers to the third (for this array, the last)
         value in the array.","The answer is not an array,
         just a boolean value. The [2] after the variable
         takes the value out.","The answer is not an array,
         just a boolean value. The [2] after the variable
         takes the value out."] ) ),

  new LearningStep(329, undefined, undefined)]
                    ) /* end LearningSteps */
                    ) /* end Action */,
new Action (
                    new Program(`
var some_var = [ false, true ];
some_var[0];

var joe = [ true ];

joe;


joe[0];

var amy = [false];
amy;
amy[0];



var x  = [true, false, true];
x;
x[0];
x[1];
x[2];


var day_is_free_0 = true;
var day_is_free_1 = false;
var day_is_free_2 = true;

var day_is_free =
    [ true, false, true];


day_is_free_0 = false;
day_is_free[0] = false;

day_is_free_0;
day_is_free[0];

day_is_free[1] = true;
day_is_free[1];
day_is_free;
day_is_free_1;


var c = [ false ];
c[1] = true;
c;
c[0];
c[1];
c[2] = false;
c;




            `
            ) /* end Program */,
                    new LearningSteps(
        /* Learning Steps */
        [
  new LearningStep(
    -330,
    "It's different to start with **0 being the first**. It's
         one of the small rules you need to remember.\n\n
    You'll get used to it with practice.\nLet's do some review.
        ",
    undefined ),

  new LearningStep(0, undefined, undefined),

  new LearningStep(1, undefined, undefined),

  new LearningStep(2, undefined, undefined),

  new LearningStep(3, undefined, undefined),
```

```
new LearningStep(4, undefined, undefined),

new LearningStep(5, undefined, undefined),

new LearningStep(6, undefined, undefined),

new LearningStep(7, undefined, undefined),

new LearningStep(8, undefined, undefined),

new LearningStep(9, undefined, undefined),

new LearningStep(10, undefined, undefined),

new LearningStep(11, undefined, undefined),

new LearningStep(12, undefined, undefined),

new LearningStep(13, undefined, undefined),

new LearningStep(14, undefined, undefined),

new LearningStep(15, undefined, undefined),

new LearningStep(16, undefined, undefined),

new LearningStep(17, undefined, undefined),

new LearningStep(18, undefined, undefined),

new LearningStep(
  19,
  "This is how to put\nthe **array value itself** on the
       stack.",
  undefined ),

new LearningStep(19, undefined, undefined),

new LearningStep(20, undefined, undefined),

new LearningStep(
  20,
  undefined,

  new Question(
    "What value is on the stack before this instruction
         executes?",
    [[],true,[true,true]],
    ["This is an empty array. The actual array is not empty."
         ,"This is the value inside the array. The variable
         joe has an array, so it puts the whole array on the
          stack.","This array has too many values in it."] )
          ),

new LearningStep(
  21,
  "This looks inside the array to get the value out.",
  undefined ),

new LearningStep(21, undefined, undefined),

new LearningStep(22, undefined, undefined),

new LearningStep(23, undefined, undefined),

new LearningStep(
  23,
  undefined,

  new Question(
    "What value is on the stack after this instruction
         executes?",
    [false,[true],[false]],
    ["","The [0] takes the value out of the array.","The
         answer is not an array, just a boolean value."] ) )
         ,

new LearningStep(24, undefined, undefined),

new LearningStep(25, undefined, undefined),

new LearningStep(26, undefined, undefined),

new LearningStep(27, undefined, undefined),

new LearningStep(28, undefined, undefined),

new LearningStep(29, undefined, undefined),

new LearningStep(30, undefined, undefined),

new LearningStep(31, undefined, undefined),

new LearningStep(32, undefined, undefined),

new LearningStep(33, undefined, undefined),

new LearningStep(34, undefined, undefined),

new LearningStep(35, undefined, undefined),

new LearningStep(36, undefined, undefined),

new LearningStep(37, undefined, undefined),

new LearningStep(
  38,
  "Another word for the number\nbetween the [ ]'s is the\n**
       index**. For example,\nfor x[0] the index is 0.\nIt's
        another word for position,\nbut it is shorter.",
  undefined ),

new LearningStep(38, undefined, undefined),

new LearningStep(39, undefined, undefined),

new LearningStep(40, undefined, undefined),

new LearningStep(41, undefined, undefined),

new LearningStep(42, undefined, undefined),

new LearningStep(43, undefined, undefined),

new LearningStep(44, undefined, undefined),

new LearningStep(45, undefined, undefined),

new LearningStep(46, undefined, undefined),

new LearningStep(47, undefined, undefined),

new LearningStep(48, undefined, undefined),

new LearningStep(49, undefined, undefined),

new LearningStep(50, undefined, undefined),

new LearningStep(51, undefined, undefined),

new LearningStep(
  51,
  undefined,

  new Question(
    "What value is on the stack after this instruction
         executes?",
    [false,[true],[false]],
    ["","The answer is not an array, just a boolean value.","
         The answer is not an array, just a boolean value."]
          ) ),

new LearningStep(52, undefined, undefined),

new LearningStep(53, undefined, undefined),

new LearningStep(54, undefined, undefined),

new LearningStep(55, undefined, undefined),

new LearningStep(
  55,
  undefined,

  new Question(
    "What value is on the stack after this instruction
         executes?",
    [true,[true],[false]],
    ["","The answer is not an array, just a boolean value.","
         The answer is not an array, just a boolean value."]
```

```
        ) ),

new LearningStep(56, undefined, undefined),

new LearningStep(57, undefined, undefined),

new LearningStep(58, undefined, undefined),

new LearningStep(59, undefined, undefined),

new LearningStep(
  59,
  undefined,

  new Question(
    "What value is on the stack after this instruction
         executes?",
    [false,[false],[true]],
    ["","The answer is not an array, just a boolean value.","
         The answer is not an array, just a boolean value."]
         ) ),

new LearningStep(60, undefined, undefined),

new LearningStep(
  61,
  "## Changing array values\n\n
  You can also **set the values inside an array**, just like
       you'd **set a variable**.\n\n
  An array is like a box\nof boxes. The positions\ninside it
       act like variables.\n\n
  In fact, an array is a lot\nlike a *namespace*, where the\
       nvariables are all named\n0 or 1 or 2 or 3 ...",
  undefined ),

new LearningStep(61, undefined, undefined),

new LearningStep(62, undefined, undefined),

new LearningStep(63, undefined, undefined),

new LearningStep(64, undefined, undefined),

new LearningStep(65, undefined, undefined),

new LearningStep(66, undefined, undefined),

new LearningStep(67, undefined, undefined),

new LearningStep(68, undefined, undefined),

new LearningStep(69, undefined, undefined),

new LearningStep(70, undefined, undefined),

new LearningStep(71, undefined, undefined),

new LearningStep(72, undefined, undefined),

new LearningStep(73, undefined, undefined),

new LearningStep(74, undefined, undefined),

new LearningStep(75, undefined, undefined),

new LearningStep(76, undefined, undefined),

new LearningStep(77, undefined, undefined),

new LearningStep(78, undefined, undefined),

new LearningStep(79, undefined, undefined),

new LearningStep(80, undefined, undefined),

new LearningStep(81, undefined, undefined),

new LearningStep(82, undefined, undefined),

new LearningStep(83, undefined, undefined),

new LearningStep(
  83,
  "These next two statements look pretty similar",
  undefined ),

new LearningStep(84, undefined, undefined),

new LearningStep(85, undefined, undefined),

new LearningStep(86, undefined, undefined),

new LearningStep(87, undefined, undefined),

new LearningStep(88, undefined, undefined),

new LearningStep(89, undefined, undefined),

new LearningStep(90, undefined, undefined),

new LearningStep(91, undefined, undefined),

new LearningStep(92, undefined, undefined),

new LearningStep(93, undefined, undefined),

new LearningStep(94, undefined, undefined),

new LearningStep(95, undefined, undefined),

new LearningStep(96, undefined, undefined),

new LearningStep(97, undefined, undefined),

new LearningStep(98, undefined, undefined),

new LearningStep(99, undefined, undefined),

new LearningStep(
  99,
  undefined,

  new Question(
    "What is the value of day_is_free after this instruction
         (not this line) executes?",
    [false,[true,false,true],[true,true,true]],
    ["This is only the first value of the array.","This array
         had its first value changed by day_is_free[0]=
         false.","This is the value it will have AFTER this
         line is done executing. This instruction just
         pushes the value on to the stack. It doesn't change
         it yet; a later instruction does that (soon)."] )
         ),

new LearningStep(100, undefined, undefined),

new LearningStep(101, undefined, undefined),

new LearningStep(102, undefined, undefined),

new LearningStep(103, undefined, undefined),

new LearningStep(104, undefined, undefined),

new LearningStep(105, undefined, undefined),

new LearningStep(106, undefined, undefined),

new LearningStep(107, undefined, undefined),

new LearningStep(
  107,
  undefined,

  new Question(
    "What is the value of day_is_free[1]?",
    [false,[false],[true]],
    ["This is the value it had before the previous line
         executed.","The answer is not an array, just a
         boolean value. The [1] takes the value out.","The
         answer is not an array, just a boolean value. The
         [1] takes the value out."] ) ),

new LearningStep(108, undefined, undefined),

new LearningStep(109, undefined, undefined),

new LearningStep(110, undefined, undefined),

new LearningStep(111, undefined, undefined),
```

```
  new LearningStep(
    112,
    "You can even put\nnew elements in this way.",
    undefined ),

  new LearningStep(112, undefined, undefined),

  new LearningStep(113, undefined, undefined),

  new LearningStep(114, undefined, undefined),

  new LearningStep(115, undefined, undefined),

  new LearningStep(116, undefined, undefined),

  new LearningStep(117, undefined, undefined),

  new LearningStep(118, undefined, undefined),

  new LearningStep(119, undefined, undefined),

  new LearningStep(120, undefined, undefined),

  new LearningStep(121, undefined, undefined),

  new LearningStep(122, undefined, undefined),

  new LearningStep(123, undefined, undefined),

  new LearningStep(124, undefined, undefined),

  new LearningStep(125, undefined, undefined),

  new LearningStep(126, undefined, undefined),

  new LearningStep(127, undefined, undefined),

  new LearningStep(128, undefined, undefined),

  new LearningStep(129, undefined, undefined),

  new LearningStep(130, undefined, undefined),

  new LearningStep(131, undefined, undefined),

  new LearningStep(132, undefined, undefined),

  new LearningStep(133, undefined, undefined),

  new LearningStep(134, undefined, undefined),

  new LearningStep(135, undefined, undefined),

  new LearningStep(136, undefined, undefined),

  new LearningStep(137, undefined, undefined),

  new LearningStep(138, undefined, undefined),

  new LearningStep(139, undefined, undefined),

  new LearningStep(
    139,
    undefined,

    new Question(
      "what is the value of c?",
      [[false],[true,false],[false,true]],
      ["This is the original value of c when it was declared.
            The lines after modified c (for example c[1]=true
            added an element).","c[1]=true changed the 2nd
            position in the array. The first position is c[0]."
            ,"This array had another value added to it by the
            line before, so this is too few elements."] ) ),

  new LearningStep(
    140,
    "Excellent! You're learning a lot!",
    undefined )]
                    ) /* end LearningSteps */
                    ) /* end Action */,
new Action (
                    new Program(`
var x = [ 40, 500 ];


x = [ 40, 500 ];
x[0];


x = [ 60, 8 ];
x[0];
x[1];

x = [ 50, 60, 70];
x[0];
x[1];
x[2];
x[1];
x[2];

x = [ 80, 90, 200];
x[2];
x[1];
x[0];


var a = [ 4 ];
a[0];


x = [ 0, 1, 2 ];
x[0];
x[1];
x[2];

x = [ 1 ];
x[0];

x = [ 2 ];
x[0];

x = [ 1, 1 ];
x[1];
x[0];


var z1 = 100;
var z2 = 1000;
var b = [z1, z2];
b[0];
b[1];
var c = [z1, z2, 20];
c[0];
c[1];
c[2];


z1 = 100;
z2 = 1000;
z3 = 10;
z4 = 1;
b = [z1, z2];
c = [z3, z4, 20];
b[0];
c[1];
c[2];
z1 = 20;
z4 = 40;
b[0];
b[1];
c[0];
c[1];
c[2];
                    `
                    ) /* end Program */,
```

```
                new LearningSteps(
      /* Learning Steps */
      [
new LearningStep(
  0,
  "## Arrays\n\n
  Arrays can also hold numbers! They can hold **any value**.
       The **positions** in an **array** work just like **
       variables** - they behave in the same way!\n\n
  Remember to step forward and\nbackwards if you're having\
       ntrouble!",
  undefined ),

new LearningStep(0, undefined, undefined),

new LearningStep(1, undefined, undefined),

new LearningStep(2, undefined, undefined),

new LearningStep(3, undefined, undefined),

new LearningStep(4, undefined, undefined),

new LearningStep(5, undefined, undefined),

new LearningStep(6, undefined, undefined),

new LearningStep(7, undefined, undefined),

new LearningStep(
  8,
  "You can read \"x[0]\" as \"the value at position 0 in x\"
       or \"the first value in the array in x\".\n\n
  This can be confusing for people, because the x[0] looks
       like a variable name next to an array with 0 inside
       it. It's not.\nBecause it follows a variable name
       immediately, the computer parser creates an
       instruction to look inside the array in the x
       variable, and find value at the 0 position.\n\n
  The language designers decided to make it work this way. It
       's short.",
  undefined ),

new LearningStep(8, undefined, undefined),

new LearningStep(9, undefined, undefined),

new LearningStep(10, undefined, undefined),

new LearningStep(11, undefined, undefined),

new LearningStep(12, undefined, undefined),

new LearningStep(13, undefined, undefined),

new LearningStep(14, undefined, undefined),

new LearningStep(15, undefined, undefined),

new LearningStep(16, undefined, undefined),

new LearningStep(17, undefined, undefined),

new LearningStep(18, undefined, undefined),

new LearningStep(19, undefined, undefined),

new LearningStep(20, undefined, undefined),

new LearningStep(21, undefined, undefined),

new LearningStep(22, undefined, undefined),

new LearningStep(23, undefined, undefined),

new LearningStep(24, undefined, undefined),

new LearningStep(25, undefined, undefined),

new LearningStep(26, undefined, undefined),

new LearningStep(27, undefined, undefined),

new LearningStep(28, undefined, undefined),

new LearningStep(
  28,
  "Another word for the number\nbetween the [ ]'s is the\
       nindex.",
  undefined ),

new LearningStep(29, undefined, undefined),

new LearningStep(30, undefined, undefined),

new LearningStep(31, undefined, undefined),

new LearningStep(32, undefined, undefined),

new LearningStep(33, undefined, undefined),

new LearningStep(34, undefined, undefined),

new LearningStep(35, undefined, undefined),

new LearningStep(36, undefined, undefined),

new LearningStep(37, undefined, undefined),

new LearningStep(38, undefined, undefined),

new LearningStep(39, undefined, undefined),

new LearningStep(40, undefined, undefined),

new LearningStep(41, undefined, undefined),

new LearningStep(42, undefined, undefined),

new LearningStep(
  42,
  "",

  new Question(
    "What value is on the stack?",
    [1,0,5],
    ["","","",""] ) ),

new LearningStep(43, undefined, undefined),

new LearningStep(44, undefined, undefined),

new LearningStep(45, undefined, undefined),

new LearningStep(46, undefined, undefined),

new LearningStep(
  46,
  "",

  new Question(
    "What value is on the stack?",
    [5,100,10],
    ["","","",""] ) ),

new LearningStep(47, undefined, undefined),

new LearningStep(48, undefined, undefined),

new LearningStep(49, undefined, undefined),

new LearningStep(50, undefined, undefined),

new LearningStep(
  50,
  "",

  new Question(
    "What value is on the stack?",
    [0,1,10],
    ["","","",""] ) ),

new LearningStep(51, undefined, undefined),

new LearningStep(52, undefined, undefined),

new LearningStep(53, undefined, undefined),

new LearningStep(54, undefined, undefined),
```

```
new LearningStep(55, undefined, undefined),

new LearningStep(56, undefined, undefined),

new LearningStep(57, undefined, undefined),

new LearningStep(58, undefined, undefined),

new LearningStep(59, undefined, undefined),

new LearningStep(60, undefined, undefined),

new LearningStep(61, undefined, undefined),

new LearningStep(62, undefined, undefined),

new LearningStep(63, undefined, undefined),

new LearningStep(64, undefined, undefined),

new LearningStep(65, undefined, undefined),

new LearningStep(66, undefined, undefined),

new LearningStep(67, undefined, undefined),

new LearningStep(68, undefined, undefined),

new LearningStep(69, undefined, undefined),

new LearningStep(
  69,
  "",

  new Question(
    "What value is on the stack?",
    [1,0,100],
    ["","",""] ) ),

new LearningStep(70, undefined, undefined),

new LearningStep(71, undefined, undefined),

new LearningStep(72, undefined, undefined),

new LearningStep(73, undefined, undefined),

new LearningStep(
  73,
  "",

  new Question(
    "What value is on the stack?",
    [10,1,0],
    ["","",""] ) ),

new LearningStep(74, undefined, undefined),

new LearningStep(75, undefined, undefined),

new LearningStep(76, undefined, undefined),

new LearningStep(77, undefined, undefined),

new LearningStep(
  77,
  "",

  new Question(
    "What value is on the stack?",
    [0,100,1],
    ["","",""] ) ),

new LearningStep(
  78,
  "Let's look at another array.\nThis one only has one
       position.",
  undefined ),

new LearningStep(78, undefined, undefined),

new LearningStep(79, undefined, undefined),

new LearningStep(80, undefined, undefined),

new LearningStep(81, undefined, undefined),

new LearningStep(82, undefined, undefined),

new LearningStep(83, undefined, undefined),

new LearningStep(84, undefined, undefined),

new LearningStep(85, undefined, undefined),

new LearningStep(86, undefined, undefined),

new LearningStep(87, undefined, undefined),

new LearningStep(88, undefined, undefined),

new LearningStep(
  89,
  "A value in an array is \ncalled an **element**. We call
       them elements because it is shorter.\n\n
  For example, you can write or say\n\"the first element\"
       instead of \n\"the value located in the first
       position\".",
  undefined ),

new LearningStep(89, undefined, undefined),

new LearningStep(90, undefined, undefined),

new LearningStep(91, undefined, undefined),

new LearningStep(92, undefined, undefined),

new LearningStep(93, undefined, undefined),

new LearningStep(94, undefined, undefined),

new LearningStep(95, undefined, undefined),

new LearningStep(96, undefined, undefined),

new LearningStep(97, undefined, undefined),

new LearningStep(98, undefined, undefined),

new LearningStep(99, undefined, undefined),

new LearningStep(100, undefined, undefined),

new LearningStep(101, undefined, undefined),

new LearningStep(102, undefined, undefined),

new LearningStep(103, undefined, undefined),

new LearningStep(104, undefined, undefined),

new LearningStep(105, undefined, undefined),

new LearningStep(106, undefined, undefined),

new LearningStep(107, undefined, undefined),

new LearningStep(108, undefined, undefined),

new LearningStep(109, undefined, undefined),

new LearningStep(110, undefined, undefined),

new LearningStep(111, undefined, undefined),

new LearningStep(112, undefined, undefined),

new LearningStep(113, undefined, undefined),

new LearningStep(114, undefined, undefined),

new LearningStep(115, undefined, undefined),

new LearningStep(116, undefined, undefined),

new LearningStep(
  116,
  undefined,
```

```
  new Question(
    "What value is on the stack?",
    [7,5,13],
    [] ) ),

new LearningStep(117, undefined, undefined),

new LearningStep(118, undefined, undefined),

new LearningStep(119, undefined, undefined),

new LearningStep(120, undefined, undefined),

new LearningStep(121, undefined, undefined),

new LearningStep(122, undefined, undefined),

new LearningStep(123, undefined, undefined),

new LearningStep(124, undefined, undefined),

new LearningStep(125, undefined, undefined),

new LearningStep(126, undefined, undefined),

new LearningStep(127, undefined, undefined),

new LearningStep(128, undefined, undefined),

new LearningStep(129, undefined, undefined),

new LearningStep(130, undefined, undefined),

new LearningStep(131, undefined, undefined),

new LearningStep(132, undefined, undefined),

new LearningStep(133, undefined, undefined),

new LearningStep(134, undefined, undefined),

new LearningStep(135, undefined, undefined),

new LearningStep(
  135,
  undefined,

  new Question(
    "What value is on the stack?",
    [0,10,20],
    [] ) ),

new LearningStep(136, undefined, undefined),

new LearningStep(137, undefined, undefined),

new LearningStep(138, undefined, undefined),

new LearningStep(139, undefined, undefined),

new LearningStep(
  140,
  "You can put any expression\nin any position in the array."
      ,
  undefined ),

new LearningStep(140, undefined, undefined),

new LearningStep(141, undefined, undefined),

new LearningStep(142, undefined, undefined),

new LearningStep(143, undefined, undefined),

new LearningStep(144, undefined, undefined),

new LearningStep(145, undefined, undefined),

new LearningStep(146, undefined, undefined),

new LearningStep(147, undefined, undefined),

new LearningStep(148, undefined, undefined),

new LearningStep(149, undefined, undefined),

new LearningStep(150, undefined, undefined),

new LearningStep(151, undefined, undefined),

new LearningStep(152, undefined, undefined),

new LearningStep(153, undefined, undefined),

new LearningStep(154, undefined, undefined),

new LearningStep(155, undefined, undefined),

new LearningStep(156, undefined, undefined),

new LearningStep(157, undefined, undefined),

new LearningStep(158, undefined, undefined),

new LearningStep(159, undefined, undefined),

new LearningStep(160, undefined, undefined),

new LearningStep(161, undefined, undefined),

new LearningStep(
  161,
  undefined,

  new Question(
    "What value is on the stack?",
    [0,10,1000],
    [] ) ),

new LearningStep(162, undefined, undefined),

new LearningStep(163, undefined, undefined),

new LearningStep(164, undefined, undefined),

new LearningStep(165, undefined, undefined),

new LearningStep(166, undefined, undefined),

new LearningStep(167, undefined, undefined),

new LearningStep(168, undefined, undefined),

new LearningStep(169, undefined, undefined),

new LearningStep(170, undefined, undefined),

new LearningStep(171, undefined, undefined),

new LearningStep(172, undefined, undefined),

new LearningStep(173, undefined, undefined),

new LearningStep(174, undefined, undefined),

new LearningStep(175, undefined, undefined),

new LearningStep(176, undefined, undefined),

new LearningStep(177, undefined, undefined),

new LearningStep(178, undefined, undefined),

new LearningStep(179, undefined, undefined),

new LearningStep(180, undefined, undefined),

new LearningStep(181, undefined, undefined),

new LearningStep(182, undefined, undefined),

new LearningStep(
  182,
  undefined,

  new Question(
    "What value is on the stack?",
    [10,100,0],
    [] ) ),
```

```
new LearningStep(183, undefined, undefined),

new LearningStep(184, undefined, undefined),

new LearningStep(185, undefined, undefined),

new LearningStep(186, undefined, undefined),

new LearningStep(
  187,
  "Let's try some questions!",
  undefined ),

new LearningStep(187, undefined, undefined),

new LearningStep(188, undefined, undefined),

new LearningStep(189, undefined, undefined),

new LearningStep(190, undefined, undefined),

new LearningStep(191, undefined, undefined),

new LearningStep(192, undefined, undefined),

new LearningStep(193, undefined, undefined),

new LearningStep(194, undefined, undefined),

new LearningStep(195, undefined, undefined),

new LearningStep(196, undefined, undefined),

new LearningStep(197, undefined, undefined),

new LearningStep(198, undefined, undefined),

new LearningStep(199, undefined, undefined),

new LearningStep(200, undefined, undefined),

new LearningStep(201, undefined, undefined),

new LearningStep(202, undefined, undefined),

new LearningStep(203, undefined, undefined),

new LearningStep(204, undefined, undefined),

new LearningStep(205, undefined, undefined),

new LearningStep(206, undefined, undefined),

new LearningStep(207, undefined, undefined),

new LearningStep(208, undefined, undefined),

new LearningStep(209, undefined, undefined),

new LearningStep(210, undefined, undefined),

new LearningStep(211, undefined, undefined),

new LearningStep(212, undefined, undefined),

new LearningStep(213, undefined, undefined),

new LearningStep(214, undefined, undefined),

new LearningStep(215, undefined, undefined),

new LearningStep(
  215,
  undefined,

  new Question(
    "What value is on the stack?",
    [0,10,1000],
    [] ) ),

new LearningStep(216, undefined, undefined),

new LearningStep(217, undefined, undefined),

new LearningStep(218, undefined, undefined),

new LearningStep(219, undefined, undefined),

new LearningStep(
  219,
  undefined,

  new Question(
    "What value is on the stack?",
    [10,100,1000],
    [] ) ),

new LearningStep(220, undefined, undefined),

new LearningStep(221, undefined, undefined),

new LearningStep(222, undefined, undefined),

new LearningStep(223, undefined, undefined),

new LearningStep(
  223,
  undefined,

  new Question(
    "What value is on the stack?",
    [10,100,1],
    ["","","c[2] is the third element. arrays start at 0."] )
          ),

new LearningStep(224, undefined, undefined),

new LearningStep(225, undefined, undefined),

new LearningStep(226, undefined, undefined),

new LearningStep(227, undefined, undefined),

new LearningStep(228, undefined, undefined),

new LearningStep(229, undefined, undefined),

new LearningStep(230, undefined, undefined),

new LearningStep(231, undefined, undefined),

new LearningStep(232, undefined, undefined),

new LearningStep(233, undefined, undefined),

new LearningStep(234, undefined, undefined),

new LearningStep(235, undefined, undefined),

new LearningStep(236, undefined, undefined),

new LearningStep(237, undefined, undefined),

new LearningStep(
  237,
  undefined,

  new Question(
    "What value is on the stack?",
    [1,10,100],
    [] ) ),

new LearningStep(238, undefined, undefined),

new LearningStep(239, undefined, undefined),

new LearningStep(240, undefined, undefined),

new LearningStep(241, undefined, undefined),

new LearningStep(
  241,
  undefined,

  new Question(
    "What value is on the stack?",
    [1,100,1000],
    [] ) ),

new LearningStep(242, undefined, undefined),
```

```
  new LearningStep(
    242,
    undefined,

    new Question(
      "What is the value of c[1]?",
      [40,100,1000],
      ["z4 did have it's value changed, but when c was created,
            what number was pushed onto the stack for that
           position?","That's the value of z1.","That's b[1]'s
            value right now."] ) ),

  new LearningStep(243, undefined, undefined),

  new LearningStep(244, undefined, undefined),

  new LearningStep(245, undefined, undefined),

  new LearningStep(246, undefined, undefined),

  new LearningStep(247, undefined, undefined),

  new LearningStep(248, undefined, undefined),

  new LearningStep(
    249,
    "Great! Programs use arrays and other ways to store
         collections of values to increase their **flexibility
         ** and **generality**, so they are essential to
         understand. Onwards!",
    undefined )]
                    ) /* end LearningSteps */
                    ) /* end Action */,
new Action (
                    new Program(`

var x = [1];
var y = [1, 10];
var z = [1, 10, 100];
x.length;
y.length;
z.length;


x = [1, 10];
x.length;
x[2] = 1000;
x.length;
x[3] = 5;
x.length;



x = [1 , 10];
y = [2, 20];

x[1];
x[0] = y[0];
x[0];
x[2] = 100;
x.length;
x[3] = y[0];
x.length;


x = [1 , 10];
z = [];
z.length;
z[0] = 5;
z.length;
z[1] = 50;
x[2]=500;
x[3]=700;
x.length;









            `
            ) /* end Program */,
                    new LearningSteps(
      /* Learning Steps */
      [
new LearningStep(
  0,
  "## Length of Arrays\n\n
  When we use an array, we don't\nalways know how many
       positions\nit will have. The number of places an
       array has is called its **length**.\n\n
  The language designers made a way to get the length of the
       array.\nYou write the name of the array followed by a
        . then the word 'length'.",
  undefined ),

new LearningStep(0, undefined, undefined),

new LearningStep(1, undefined, undefined),

new LearningStep(2, undefined, undefined),

new LearningStep(3, undefined, undefined),

new LearningStep(4, undefined, undefined),

new LearningStep(5, undefined, undefined),

new LearningStep(6, undefined, undefined),

new LearningStep(7, undefined, undefined),

new LearningStep(8, undefined, undefined),

new LearningStep(9, undefined, undefined),

new LearningStep(10, undefined, undefined),

new LearningStep(11, undefined, undefined),

new LearningStep(12, undefined, undefined),

new LearningStep(13, undefined, undefined),

new LearningStep(14, undefined, undefined),

new LearningStep(15, undefined, undefined),

new LearningStep(16, undefined, undefined),

new LearningStep(17, undefined, undefined),

new LearningStep(18, undefined, undefined),

new LearningStep(19, undefined, undefined),

new LearningStep(20, undefined, undefined),

new LearningStep(21, undefined, undefined),

new LearningStep(22, undefined, undefined),

new LearningStep(23, undefined, undefined),

new LearningStep(
  23,
  "Watch how .length works",
  undefined ),

new LearningStep(24, undefined, undefined),

new LearningStep(25, undefined, undefined),

new LearningStep(26, undefined, undefined),

new LearningStep(27, undefined, undefined),

new LearningStep(28, undefined, undefined),

new LearningStep(29, undefined, undefined),

new LearningStep(30, undefined, undefined),

new LearningStep(31, undefined, undefined),

new LearningStep(
  32,
  undefined,
```

```
  new Question(
    "What is the length of the array in the z variable?",
    [0,100,10],
    ["The array is not empty. Look at its declaration to see
         how many positions it has.","Too many.","Too many."
         ] ) ),

new LearningStep(32, undefined, undefined),

new LearningStep(
  33,
  "The number of positions in an array can change in
       Javascript. This makes arrays more flexible.",
  undefined ),

new LearningStep(33, undefined, undefined),

new LearningStep(34, undefined, undefined),

new LearningStep(35, undefined, undefined),

new LearningStep(36, undefined, undefined),

new LearningStep(37, undefined, undefined),

new LearningStep(38, undefined, undefined),

new LearningStep(39, undefined, undefined),

new LearningStep(40, undefined, undefined),

new LearningStep(41, undefined, undefined),

new LearningStep(42, undefined, undefined),

new LearningStep(43, undefined, undefined),

new LearningStep(44, undefined, undefined),

new LearningStep(45, undefined, undefined),

new LearningStep(46, undefined, undefined),

new LearningStep(47, undefined, undefined),

new LearningStep(48, undefined, undefined),

new LearningStep(49, undefined, undefined),

new LearningStep(50, undefined, undefined),

new LearningStep(51, undefined, undefined),

new LearningStep(52, undefined, undefined),

new LearningStep(53, undefined, undefined),

new LearningStep(54, undefined, undefined),

new LearningStep(55, undefined, undefined),

new LearningStep(56, undefined, undefined),

new LearningStep(57, undefined, undefined),

new LearningStep(
  57,
  undefined,

  new Question(
    "What is the length of the array in the x variable?",
    [100,5,1],
    ["","",""] ) ),

new LearningStep(
  58,
  "Let's try some questions!",
  undefined ),

new LearningStep(58, undefined, undefined),

new LearningStep(59, undefined, undefined),

new LearningStep(60, undefined, undefined),

new LearningStep(61, undefined, undefined),

new LearningStep(62, undefined, undefined),

new LearningStep(63, undefined, undefined),

new LearningStep(64, undefined, undefined),

new LearningStep(65, undefined, undefined),

new LearningStep(66, undefined, undefined),

new LearningStep(67, undefined, undefined),

new LearningStep(68, undefined, undefined),

new LearningStep(69, undefined, undefined),

new LearningStep(70, undefined, undefined),

new LearningStep(71, undefined, undefined),

new LearningStep(72, undefined, undefined),

new LearningStep(73, undefined, undefined),

new LearningStep(
  73,
  undefined,

  new Question(
    "What is the value of x[1]?",
    [1,2,20],
    ["x[1] is the value in the second position of the array.
         x[0] is the first.","2 is a value inside y. The
         answer is inside an array though!","20 is y[1].
         This is asking for x[1]."] ) ),

new LearningStep(74, undefined, undefined),

new LearningStep(75, undefined, undefined),

new LearningStep(76, undefined, undefined),

new LearningStep(77, undefined, undefined),

new LearningStep(78, undefined, undefined),

new LearningStep(79, undefined, undefined),

new LearningStep(80, undefined, undefined),

new LearningStep(81, undefined, undefined),

new LearningStep(82, undefined, undefined),

new LearningStep(83, undefined, undefined),

new LearningStep(84, undefined, undefined),

new LearningStep(
  84,
  undefined,

  new Question(
    "What is on the stack?",
    [1,10,20],
    ["x[0] got changed. y[0] was pushed on the stack and that
         value became x[0]'s on the previous line. Close!",
         "10 is x[1]. Right idea though!","20 is y[1]. You'
         re right that the value got set to one of y's
         values!"] ) ),

new LearningStep(85, undefined, undefined),

new LearningStep(86, undefined, undefined),

new LearningStep(87, undefined, undefined),

new LearningStep(88, undefined, undefined),

new LearningStep(89, undefined, undefined),

new LearningStep(90, undefined, undefined),
```

```
new LearningStep(91, undefined, undefined),

new LearningStep(92, undefined, undefined),

new LearningStep(
  92,
  undefined,

  new Question(
    "What is the length of the array in x?",
    [2,4,1],
    ["This is how many positions x had when it was
         initialized. Go back and watch what the previous
         line did.","x[2]=100; only added 100 to the array.
         Look at the namespace, how many positions does x
         have?","This gets the length of the array stored in
          x. Look at it in the namespace, how many positions
          does it have?"] ) ),

new LearningStep(93, undefined, undefined),

new LearningStep(94, undefined, undefined),

new LearningStep(95, undefined, undefined),

new LearningStep(96, undefined, undefined),

new LearningStep(97, undefined, undefined),

new LearningStep(98, undefined, undefined),

new LearningStep(99, undefined, undefined),

new LearningStep(100, undefined, undefined),

new LearningStep(101, undefined, undefined),

new LearningStep(102, undefined, undefined),

new LearningStep(
  102,
  undefined,

  new Question(
    "What value is on the stack?",
    [2,3,1],
    ["This is how many positions x had when it was
         initialized. Go back and watch what the previous
         line did.","x[2]=100; added 100 to the array. What
         did x[3] = y[0] do?","The length of x got pushed
         onto the array. Look at x in the namespace, how
         many positions does it have?"] ) ),

new LearningStep(103, undefined, undefined),

new LearningStep(104, undefined, undefined),

new LearningStep(105, undefined, undefined),

new LearningStep(106, undefined, undefined),

new LearningStep(107, undefined, undefined),

new LearningStep(108, undefined, undefined),

new LearningStep(109, undefined, undefined),

new LearningStep(110, undefined, undefined),

new LearningStep(111, undefined, undefined),

new LearningStep(112, undefined, undefined),

new LearningStep(113, undefined, undefined),

new LearningStep(114, undefined, undefined),

new LearningStep(115, undefined, undefined),

new LearningStep(
  115,
  undefined,

  new Question(
    "What value is on the stack?",
    [3,2,1],
    ["z doesn't have any elements yet.","z doesn't have any
         elements yet.","An empty array has 0 length."] ) ),

new LearningStep(116, undefined, undefined),

new LearningStep(117, undefined, undefined),

new LearningStep(118, undefined, undefined),

new LearningStep(119, undefined, undefined),

new LearningStep(120, undefined, undefined),

new LearningStep(121, undefined, undefined),

new LearningStep(122, undefined, undefined),

new LearningStep(123, undefined, undefined),

new LearningStep(
  123,
  undefined,

  new Question(
    "What value is on the stack?",
    [3,2,0],
    ["z only holds values for it's own array. The length of z
         plus the lenght of x is 3.","How many elements did
         z[0] = 5 add?","You're right that z used to have
        no elements, but what did z[0] = 5 do?"] ) ),

new LearningStep(124, undefined, undefined),

new LearningStep(125, undefined, undefined),

new LearningStep(126, undefined, undefined),

new LearningStep(127, undefined, undefined),

new LearningStep(128, undefined, undefined),

new LearningStep(129, undefined, undefined),

new LearningStep(130, undefined, undefined),

new LearningStep(131, undefined, undefined),

new LearningStep(132, undefined, undefined),

new LearningStep(133, undefined, undefined),

new LearningStep(134, undefined, undefined),

new LearningStep(135, undefined, undefined),

new LearningStep(136, undefined, undefined),

new LearningStep(137, undefined, undefined),

new LearningStep(138, undefined, undefined),

new LearningStep(139, undefined, undefined),

new LearningStep(140, undefined, undefined),

new LearningStep(141, undefined, undefined),

new LearningStep(
  141,
  undefined,

  new Question(
    "What value is on the stack?",
    [2,3,0],
    ["x started out with only 2 places. When x[2] was
         assigned a value, it increased in size. Look at all
          the statements with x in it. How did x change?","x
         [3] is the 4th position in x. x[0] is the first
         position. You're close!","x is not an empty array.
         Look in the namespace to see x."] ) )]
                ) /* end LearningSteps */
                ) /* end Action */,
new Action (
```

```
                    new Program(`

var bedroom_length = 12;
var bedroom_width = 10;
var bedroom_height = 8;

var livingroom_length = 20;
var livingroom_width = 12;
var livingroom_height = 8;



            `
            ) /* end Program */,
                    new LearningSteps(
        /* Learning Steps */
        [
  new LearningStep(
    0,
    "## Advice Program\n\n
    Let's use what we've learned so far to start making a
          program that analyzes a house or apartment to
          recommend what air conditioner to buy.\n\n
    Different air conditioners can handle different total
          volumes (volume is the size of the space). More
          expensive ones can handle larger volumes (they have
          to cool more air so they cost more).\n\n
    Our program will eventually calculate the total volume of
          your house for you, then recommend an air conditioner
          .",
    undefined ),

  new LearningStep(
    0,
    "We'll start showing you the \nbeginning of that \nprogram,
           using what you\nhave learned!\n\n
    We'll start with imagining\nthere are two rooms in\nin the
          house.",
    undefined ),

  new LearningStep(0, undefined, undefined),

  new LearningStep(1, undefined, undefined),

  new LearningStep(2, undefined, undefined),

  new LearningStep(3, undefined, undefined),

  new LearningStep(4, undefined, undefined),

  new LearningStep(5, undefined, undefined),

  new LearningStep(6, undefined, undefined),

  new LearningStep(7, undefined, undefined),

  new LearningStep(8, undefined, undefined),

  new LearningStep(9, undefined, undefined),

  new LearningStep(10, undefined, undefined),

  new LearningStep(11, undefined, undefined),

  new LearningStep(12, undefined, undefined),

  new LearningStep(13, undefined, undefined),

  new LearningStep(14, undefined, undefined),

  new LearningStep(15, undefined, undefined),

  new LearningStep(16, undefined, undefined),

  new LearningStep(17, undefined, undefined),

  new LearningStep(18, undefined, undefined),

  new LearningStep(19, undefined, undefined),

  new LearningStep(20, undefined, undefined),

  new LearningStep(21, undefined, undefined),

  new LearningStep(22, undefined, undefined),

  new LearningStep(23, undefined, undefined),

  new LearningStep(24, undefined, undefined),

  new LearningStep(25, undefined, undefined),

  new LearningStep(26, undefined, undefined),

  new LearningStep(27, undefined, undefined),

  new LearningStep(28, undefined, undefined),

  new LearningStep(29, undefined, undefined),

  new LearningStep(
    29,
    "The next parts of the program\nwould calculate the volume
          .\nWe'll see how to do that\nsoon. ",
    undefined ),

  new LearningStep(
    29,
    "First, let's think if there\nis a better way to write\
          nthis program. You can imagine\nif there were 10
          rooms, we would start having a lot of variables and
          the program would start getting very long.\n\n
    If someone else wanted to\nuse the program for their house
          ,\nthey'd need to add more\nvariables if their house\
          nhad more rooms. Or remove\nparts of the program if\
          nthey have fewer rooms.",
    undefined ),

  new LearningStep(
    29,
    "Think to yourself and answer: What language feature might
          help here?",
    undefined )]
                    ) /* end LearningSteps */
                    ) /* end Action */,
new Action (
                    new Program(`


var bedroom_length = 12;
var bedroom_width = 10;
var bedroom_height = 8;

var bedroom = [ 12, 10, 8 ];

var livingroom_length = 20;
var livingroom_width = 12;
var livingroom_height = 8;

var living_room = [ 20, 12, 8];

var bedroom_length = 12;
var bedroom_width = 10;
var bedroom_height = 8;
var livingroom_length = 20;
var livingroom_width = 12;
var livingroom_height = 8;


var bedroom = [ 12, 10, 8 ];
var living_room = [ 20, 12, 8];


var rooms = [ [12, 10, 8],
              [20, 12, 8]];









            `
            ) /* end Program */,
                    new LearningSteps(
        /* Learning Steps */
        [
  new LearningStep(
```

```
  0,
  "Using **arrays**, we can\nmake the program more flexible\
      nand shorter.\n\n
  We'll repeat the earlier\ncode here so it's easier\nto look
      at and understand\nthe next part.",
  undefined ),

new LearningStep(0, undefined, undefined),

new LearningStep(1, undefined, undefined),

new LearningStep(2, undefined, undefined),

new LearningStep(3, undefined, undefined),

new LearningStep(4, undefined, undefined),

new LearningStep(5, undefined, undefined),

new LearningStep(6, undefined, undefined),

new LearningStep(7, undefined, undefined),

new LearningStep(8, undefined, undefined),

new LearningStep(9, undefined, undefined),

new LearningStep(10, undefined, undefined),

new LearningStep(11, undefined, undefined),

new LearningStep(12, undefined, undefined),

new LearningStep(13, undefined, undefined),

new LearningStep(14, undefined, undefined),

new LearningStep(
  14,
  "Let's use an array instead. We'll put the length first,
      width second, and height third.",
  undefined ),

new LearningStep(15, undefined, undefined),

new LearningStep(16, undefined, undefined),

new LearningStep(17, undefined, undefined),

new LearningStep(18, undefined, undefined),

new LearningStep(19, undefined, undefined),

new LearningStep(20, undefined, undefined),

new LearningStep(21, undefined, undefined),

new LearningStep(22, undefined, undefined),

new LearningStep(23, undefined, undefined),

new LearningStep(24, undefined, undefined),

new LearningStep(25, undefined, undefined),

new LearningStep(26, undefined, undefined),

new LearningStep(27, undefined, undefined),

new LearningStep(28, undefined, undefined),

new LearningStep(29, undefined, undefined),

new LearningStep(30, undefined, undefined),

new LearningStep(31, undefined, undefined),

new LearningStep(32, undefined, undefined),

new LearningStep(33, undefined, undefined),

new LearningStep(34, undefined, undefined),

new LearningStep(35, undefined, undefined),

new LearningStep(36, undefined, undefined),

new LearningStep(37, undefined, undefined),

new LearningStep(38, undefined, undefined),

new LearningStep(39, undefined, undefined),

new LearningStep(40, undefined, undefined),

new LearningStep(41, undefined, undefined),

new LearningStep(42, undefined, undefined),

new LearningStep(43, undefined, undefined),

new LearningStep(44, undefined, undefined),

new LearningStep(45, undefined, undefined),

new LearningStep(46, undefined, undefined),

new LearningStep(47, undefined, undefined),

new LearningStep(
  47,
  "Let's compare the two ways of storing room information.",
  undefined ),

new LearningStep(48, undefined, undefined),

new LearningStep(49, undefined, undefined),

new LearningStep(50, undefined, undefined),

new LearningStep(51, undefined, undefined),

new LearningStep(52, undefined, undefined),

new LearningStep(53, undefined, undefined),

new LearningStep(54, undefined, undefined),

new LearningStep(55, undefined, undefined),

new LearningStep(56, undefined, undefined),

new LearningStep(57, undefined, undefined),

new LearningStep(58, undefined, undefined),

new LearningStep(59, undefined, undefined),

new LearningStep(60, undefined, undefined),

new LearningStep(61, undefined, undefined),

new LearningStep(62, undefined, undefined),

new LearningStep(63, undefined, undefined),

new LearningStep(64, undefined, undefined),

new LearningStep(65, undefined, undefined),

new LearningStep(66, undefined, undefined),

new LearningStep(67, undefined, undefined),

new LearningStep(68, undefined, undefined),

new LearningStep(69, undefined, undefined),

new LearningStep(70, undefined, undefined),

new LearningStep(71, undefined, undefined),

new LearningStep(72, undefined, undefined),

new LearningStep(73, undefined, undefined),

new LearningStep(74, undefined, undefined),

new LearningStep(75, undefined, undefined),
```

```
new LearningStep(76, undefined, undefined),

new LearningStep(77, undefined, undefined),

new LearningStep(78, undefined, undefined),

new LearningStep(79, undefined, undefined),

new LearningStep(80, undefined, undefined),

new LearningStep(81, undefined, undefined),

new LearningStep(82, undefined, undefined),

new LearningStep(83, undefined, undefined),

new LearningStep(84, undefined, undefined),

new LearningStep(85, undefined, undefined),

new LearningStep(86, undefined, undefined),

new LearningStep(87, undefined, undefined),

new LearningStep(88, undefined, undefined),

new LearningStep(89, undefined, undefined),

new LearningStep(90, undefined, undefined),

new LearningStep(91, undefined, undefined),

new LearningStep(92, undefined, undefined),

new LearningStep(93, undefined, undefined),

new LearningStep(94, undefined, undefined),

new LearningStep(95, undefined, undefined),

new LearningStep(
  95,
  "This is shorter, but we'd still need to add and remove
       variables if there were a different number of rooms.\
       n\n
  Let's put all the rooms into a single array.",
  undefined ),

new LearningStep(96, undefined, undefined),

new LearningStep(97, undefined, undefined),

new LearningStep(98, undefined, undefined),

new LearningStep(99, undefined, undefined),

new LearningStep(100, undefined, undefined),

new LearningStep(101, undefined, undefined),

new LearningStep(102, undefined, undefined),

new LearningStep(103, undefined, undefined),

new LearningStep(104, undefined, undefined),

new LearningStep(105, undefined, undefined),

new LearningStep(106, undefined, undefined),

new LearningStep(107, undefined, undefined),

new LearningStep(108, undefined, undefined),

new LearningStep(108, undefined, undefined),

new LearningStep(109, undefined, undefined),

new LearningStep(110, undefined, undefined),

new LearningStep(111, undefined, undefined),

new LearningStep(
  111,
  "This is **even shorter**, and there can be as many rooms
       as you want in there. If you want to add a room, you
       can add a comma after the last array, then add
       another one.\n\n
  \n",
  undefined ),

new LearningStep(
  111,
  "Arrays help make programs **flexible**. It can take a lot
       of time to write a program and check it is correct.
       We can save others time and frustration by writing in
        a way that others can use our programs with a few
       changes.\n\n
  We can also save ourselves a lot of time by adapting what
       others have already written; this is called **reusing
       ** code.\n\n
  We'll learn how to multiply the room's dimensions together
       to get the volume after reviewing what we've learned!
       ",
  undefined )]
                    ) /* end LearningSteps */
                    ) /* end Action */]),
new KnowledgeUnit("Operators + - * / ( )",
          "Operators do a calculation of some kind.
They use values on the stack, then
put a value on top of the stack.
They work by using an instruction
built into the language. ",
           [
new Action (
                    new Program(`


-1;
-20;



1 + 2;
10 + 11;


1 - 2;
1 - 1;
0 - 1;



1 + 2;
1 + -2;



var y = 10 + 100;




          `
          ) /* end Program */,
                    new LearningSteps(
      /* Learning Steps */
      [
  new LearningStep(
    0,
    "## Operators\nWe know how to store values using **
         variables** and **arrays**. How do we use those
         values?\n",
    undefined ),

  new LearningStep(
    0,
    "**Operators** do a calculation of some kind using the **
         stack**. For example, the operator **+** takes two
         numbers and adds them.\n\n
    They all work the same way. They\n1) take values **off**
         the stack\n2) use those values to do a calculation\n3
         ) put a value **on top** of the stack\n\n
    They work by using an **instruction** built into the
         language.",
    undefined ),

  new LearningStep(
    0,
    "## Unary Operators\n\n
```

```
  Unary operators only act on one value. Unary means related
       to 1.\n\n
  For example, the **-** when next to a number takes its
       negative. \n\n
  Read the instruction descriptions and watch the stack to
       see how it works.",
  undefined ),

new LearningStep(0, undefined, undefined),

new LearningStep(1, undefined, undefined),

new LearningStep(2, undefined, undefined),

new LearningStep(3, undefined, undefined),

new LearningStep(4, undefined, undefined),

new LearningStep(5, undefined, undefined),

new LearningStep(
  5,
  undefined,

  new Question(
    "What value is on the stack?",
    [5,0,100],
    ["","",""] ) ),

new LearningStep(
  6,
  "## Binary Operators\n\n
  The **+** operator adds two numbers. It takes the
       expression on its left and the expression on its
       right and adds them together.\n\n
  Since it takes these 2 expressions and uses them, it is
       called a binary operator (binary means based on 2).",
  undefined ),

new LearningStep(6, undefined, undefined),

new LearningStep(7, undefined, undefined),

new LearningStep(8, undefined, undefined),

new LearningStep(9, undefined, undefined),

new LearningStep(
  9,
  undefined,

  new Question(
    "What value is on the stack?",
    [0,5,1],
    ["","",""] ) ),

new LearningStep(10, undefined, undefined),

new LearningStep(11, undefined, undefined),

new LearningStep(12, undefined, undefined),

new LearningStep(13, undefined, undefined),

new LearningStep(
  13,
  undefined,

  new Question(
    "What value is on the stack?",
    [10,1,5],
    ["","",""] ) ),

new LearningStep(
  14,
  "The **-** is another one; it subtracts.",
  undefined ),

new LearningStep(14, undefined, undefined),

new LearningStep(15, undefined, undefined),

new LearningStep(16, undefined, undefined),

new LearningStep(17, undefined, undefined),

new LearningStep(
  17,
  undefined,

  new Question(
    "What value is on the stack?",
    [5,100,10],
    ["","",""] ) ),

new LearningStep(18, undefined, undefined),

new LearningStep(19, undefined, undefined),

new LearningStep(20, undefined, undefined),

new LearningStep(21, undefined, undefined),

new LearningStep(
  21,
  undefined,

  new Question(
    "What value is on the stack?",
    [1,5,100],
    ["","",""] ) ),

new LearningStep(22, undefined, undefined),

new LearningStep(23, undefined, undefined),

new LearningStep(24, undefined, undefined),

new LearningStep(25, undefined, undefined),

new LearningStep(
  25,
  undefined,

  new Question(
    "What value is on the stack?",
    [10,1,0],
    ["","",""] ) ),

new LearningStep(
  26,
  "Let's see how we can use binary and unary operators
       together.",
  undefined ),

new LearningStep(26, undefined, undefined),

new LearningStep(27, undefined, undefined),

new LearningStep(28, undefined, undefined),

new LearningStep(29, undefined, undefined),

new LearningStep(
  29,
  undefined,

  new Question(
    "What value is on the stack?",
    [5,10,100],
    ["","",""] ) ),

new LearningStep(30, undefined, undefined),

new LearningStep(31, undefined, undefined),

new LearningStep(32, undefined, undefined),

new LearningStep(33, undefined, undefined),

new LearningStep(34, undefined, undefined),

new LearningStep(
  34,
  undefined,

  new Question(
    "What value is on the stack?",
    [1,10,100],
    ["","",""] ) ),
```

```
  new LearningStep(
    35,
    "You can set a variable's value to  be any expression. That
          expression\n gets computed once, pushed onto the\n
         stack, and then the value gets\n stored in the
         variable.",
    undefined ),

  new LearningStep(35, undefined, undefined),

  new LearningStep(36, undefined, undefined),

  new LearningStep(37, undefined, undefined),

  new LearningStep(38, undefined, undefined),

  new LearningStep(39, undefined, undefined),

  new LearningStep(40, undefined, undefined),

  new LearningStep(
    40,
    undefined,

    new Question(
      "What is the value of y?",
      [0,10,100],
      ["","10 was pushed onto the stack but it was added to
           100.","100 was pushed onto the stack but it was
           added to 10."] ) ),

  new LearningStep(41, undefined, undefined)]
                    ) /* end LearningSteps */
                    ) /* end Action */,
new Action (
                    new Program(`

1 * 2;
0 * 1;


2 / 1;
10 / 5;
1 / 2;



var x = 1 / 2;
var y = -1 / 2;




            `
            ) /* end Program */,
                    new LearningSteps(
        /* Learning Steps */
        [
  new LearningStep(
    0,
    "## Multiply and Divide Operators \n\n
    The * is another binary operator; it multiplies.",
    undefined ),

  new LearningStep(0, undefined, undefined),

  new LearningStep(1, undefined, undefined),

  new LearningStep(2, undefined, undefined),

  new LearningStep(3, undefined, undefined),

  new LearningStep(
    3,
    undefined,

    new Question(
      "What value is on the stack?",
      [5,100,1],
      ["","",""] ) ),

  new LearningStep(4, undefined, undefined),

  new LearningStep(5, undefined, undefined),

  new LearningStep(6, undefined, undefined),

  new LearningStep(7, undefined, undefined),

  new LearningStep(
    7,
    undefined,

    new Question(
      "What value is on the stack?",
      [100,5,10],
      ["","",""] ) ),

  new LearningStep(
    8,
    "The / is a binary operator; it divides.",
    undefined ),

  new LearningStep(8, undefined, undefined),

  new LearningStep(9, undefined, undefined),

  new LearningStep(10, undefined, undefined),

  new LearningStep(11, undefined, undefined),

  new LearningStep(
    11,
    undefined,

    new Question(
      "What value is on the stack?",
      [1,10,0],
      ["","",""] ) ),

  new LearningStep(12, undefined, undefined),

  new LearningStep(13, undefined, undefined),

  new LearningStep(14, undefined, undefined),

  new LearningStep(15, undefined, undefined),

  new LearningStep(
    15,
    undefined,

    new Question(
      "What value is on the stack?",
      [0,100,1],
      ["","",""] ) ),

  new LearningStep(16, undefined, undefined),

  new LearningStep(17, undefined, undefined),

  new LearningStep(18, undefined, undefined),

  new LearningStep(19, undefined, undefined),

  new LearningStep(
    19,
    undefined,

    new Question(
      "What value is on the stack?",
      [5,10,0],
      ["","",""] ) ),

  new LearningStep(
    20,
    "Let's use operators together. Watch how they use the **
         stack** to work together.",
    undefined ),

  new LearningStep(20, undefined, undefined),

  new LearningStep(21, undefined, undefined),

  new LearningStep(22, undefined, undefined),

  new LearningStep(23, undefined, undefined),
```

```
  new LearningStep(24, undefined, undefined),

  new LearningStep(25, undefined, undefined),

  new LearningStep(
    25,
    undefined,

    new Question(
      "What is the value of x?",
      [100,5,10],
      ["","",""] ) ),

  new LearningStep(26, undefined, undefined),

  new LearningStep(27, undefined, undefined),

  new LearningStep(28, undefined, undefined),

  new LearningStep(29, undefined, undefined),

  new LearningStep(30, undefined, undefined),

  new LearningStep(31, undefined, undefined),

  new LearningStep(32, undefined, undefined),

  new LearningStep(33, undefined, undefined),

  new LearningStep(
    33,
    undefined,

    new Question(
      "What is the value of y?",
      [1,100,10],
      ["","",""] ) ),

  new LearningStep(34, undefined, undefined)]
                    ) /* end LearningSteps */
                    ) /* end Action */,
new Action (
                    new Program(`

100 + (1 + 10);

1 - (2 * 3);
1 - 2 * 3;
(1 - 2) * 3;

1 + (2 * (3 - 1));
1 + ((2 + 3) - 1);

(3 - 1) * 2;
(3 + 2) * 4;
(3 + (1 * 2)) - 4;




            `
            ) /* end Program */,
                    new LearningSteps(
        /* Learning Steps */
        [
  new LearningStep(
    0,
    "## Order of Operators\n\n
    When there are many operators in an expression, in what
        order will the computer execute them?\n\n
    The computer uses a set of rules to determine which
        operators to do first, second, ...\n\n
    These rules are called precedence rules (which operators
        should precede each other). In Javascript and most
        programming languages the rules are like traditional
        math notation. \n\n
    You don't have to remember all the rules, but do remember
        that expressions in parentheses ( ) get calculated
        first. \n\n
    \n\n
    \n\n
    ",
    undefined ),

  new LearningStep(0, undefined, undefined),

  new LearningStep(1, undefined, undefined),

  new LearningStep(2, undefined, undefined),

  new LearningStep(3, undefined, undefined),

  new LearningStep(4, undefined, undefined),

  new LearningStep(5, undefined, undefined),

  new LearningStep(
    5,
    "Sometimes, the order doesn't make a difference.\n\n
    For example, if you add 100 to 1 and get 101 and then add
        10, you get 111. \n\n
    If you add 1 and 10 to get 11, then add 11 to 100, you
        still get 111. \n\n
    The order you add numbers together doesn't change the
        result.",
    undefined ),

  new LearningStep(
    5,
    "Sometimes, the order does make a difference.\n\n
    Watch these next lines and see how the order makes a
        difference.",
    undefined ),

  new LearningStep(6, undefined, undefined),

  new LearningStep(7, undefined, undefined),

  new LearningStep(8, undefined, undefined),

  new LearningStep(9, undefined, undefined),

  new LearningStep(10, undefined, undefined),

  new LearningStep(11, undefined, undefined),

  new LearningStep(
    11,
    undefined,

    new Question(
      "What value is on the stack?",
      [1,5,100],
      ["","",""] ) ),

  new LearningStep(12, undefined, undefined),

  new LearningStep(13, undefined, undefined),

  new LearningStep(14, undefined, undefined),

  new LearningStep(15, undefined, undefined),

  new LearningStep(16, undefined, undefined),

  new LearningStep(17, undefined, undefined),

  new LearningStep(
    17,
    undefined,

    new Question(
      "What value is on the stack?",
      [0,100,1],
      ["","",""] ) ),

  new LearningStep(18, undefined, undefined),

  new LearningStep(19, undefined, undefined),

  new LearningStep(20, undefined, undefined),

  new LearningStep(21, undefined, undefined),

  new LearningStep(22, undefined, undefined),

  new LearningStep(23, undefined, undefined),

  new LearningStep(
```

```
  23,
  undefined,

  new Question(
    "What value is on the stack?",
    [0,10,5],
    ["","",""] ) ),

new LearningStep(
  23,
  "In general, whenever the order is unclear, write
       parentheses in your programs to make it explicit.
       This makes your programs **easier for people to read
       **. It also helps you make fewer errors. \n\n
  Let's watch how the inner-most parentheses get executed
       first.",
  undefined ),

new LearningStep(24, undefined, undefined),

new LearningStep(25, undefined, undefined),

new LearningStep(26, undefined, undefined),

new LearningStep(27, undefined, undefined),

new LearningStep(28, undefined, undefined),

new LearningStep(29, undefined, undefined),

new LearningStep(30, undefined, undefined),

new LearningStep(31, undefined, undefined),

new LearningStep(
  31,
  undefined,

  new Question(
    "What value is on the stack?",
    [1,0,100],
    ["","",""] ) ),

new LearningStep(32, undefined, undefined),

new LearningStep(33, undefined, undefined),

new LearningStep(34, undefined, undefined),

new LearningStep(35, undefined, undefined),

new LearningStep(36, undefined, undefined),

new LearningStep(37, undefined, undefined),

new LearningStep(38, undefined, undefined),

new LearningStep(39, undefined, undefined),

new LearningStep(
  39,
  undefined,

  new Question(
    "What value is on the stack?",
    [100,0,10],
    ["","",""] ) ),

new LearningStep(40, undefined, undefined),

new LearningStep(41, undefined, undefined),

new LearningStep(42, undefined, undefined),

new LearningStep(43, undefined, undefined),

new LearningStep(44, undefined, undefined),

new LearningStep(45, undefined, undefined),

new LearningStep(
  45,
  undefined,

  new Question(
    "What value is on the stack?",
    [1,10,5],
    ["","",""] ) ),

new LearningStep(46, undefined, undefined),

new LearningStep(47, undefined, undefined),

new LearningStep(48, undefined, undefined),

new LearningStep(49, undefined, undefined),

new LearningStep(50, undefined, undefined),

new LearningStep(51, undefined, undefined),

new LearningStep(
  51,
  undefined,

  new Question(
    "What value is on the stack?",
    [5,0,100],
    ["","",""] ) ),

new LearningStep(52, undefined, undefined),

new LearningStep(53, undefined, undefined),

new LearningStep(54, undefined, undefined),

new LearningStep(55, undefined, undefined),

new LearningStep(56, undefined, undefined),

new LearningStep(57, undefined, undefined),

new LearningStep(58, undefined, undefined),

new LearningStep(59, undefined, undefined),

new LearningStep(
  59,
  undefined,

  new Question(
    "What value is on the stack?",
    [10,0,5],
    ["","",""] ) ),

new LearningStep(
  59,
  "If this point seems confusing,\ndo not worry about it; a
       few people have this concern.\n\n
  The language designers have created constraints so that you
       can **never** have an expression that ends up
       leaving more than one value on the stack.\n\n
  We're telling you this because it means you **never** have
       to worry about some parentheses\nleaving extra things
       on the stack, or using up a value on the stack that
       causes an error. \n\n
  The parser, while it looks at all the letters in a program
       and generates instructions from it, will recognize
       that problem and tell the programmer there is an
       error. ",
  undefined ),

new LearningStep(59, undefined, undefined)]
                  ) /* end LearningSteps */
                  ) /* end Action */]),
new KnowledgeUnit("Comparison Operators == != > >= < <=",
          "Operators use values on the stack to do a
               calculation. The not operator (!) changes a
               true to false and changes false to true. The
               == operator compares two values and puts true
                on the stack if they are equal; otherwise it
                puts false.Comparison operators can use
               number values or boolean values. They
               evaluate to either true or false. + - * / > <
                >= <= == != work on numbers. ! != == && ||
               work on booleans.",
            [
new Action (
                    new Program(`
```

```
var secret = 8;
var guess = 7;

secret == guess;

var are_they_equal = secret == guess;


8 == 9;
10 == 10;
10302 == 10302;
4 == 0;
1 == -1;
0 == 0;


true == true;
false == false;


false == true;
true == false;



true == (false == false);
(true == false) == false;
(true == false) == true;


false != true;
true  != false;
true != true;
1 != 0;




            `
            ) /* end Program */,
                    new LearningSteps(
        /* Learning Steps */
        [
  new LearningStep(
    0,
    "## Equality Operator\n\n
    Let's imagine a guessing game program.\nThere's a secret
          number from one to ten, and a person enters a guess.
          We want to tell if the guess is the same as the
          secret number.\n\n
    What's the computer code to do this?\n\n
    The **===** can help us. It is another operator. It uses
          values on the stack, then puts a value on top of the
          stack.\n\n
    The **===** operator compares the expression to its left\
          nand the expression to its right.\n\n
    If they are equal, it puts true on the stack.\n\n
    If they are not equal, it puts false\non the stack.\n\n
    Let's watch it work!",
    undefined ),

  new LearningStep(0, undefined, undefined),

  new LearningStep(1, undefined, undefined),

  new LearningStep(2, undefined, undefined),

  new LearningStep(3, undefined, undefined),

  new LearningStep(4, undefined, undefined),

  new LearningStep(5, undefined, undefined),

  new LearningStep(6, undefined, undefined),

  new LearningStep(7, undefined, undefined),

  new LearningStep(8, undefined, undefined),

  new LearningStep(9, undefined, undefined),

  new LearningStep(10, undefined, undefined),

  new LearningStep(11, undefined, undefined),

  new LearningStep(12, undefined, undefined),

  new LearningStep(13, undefined, undefined),

  new LearningStep(
    13,
    "8 doesn't equal 7, so the == operator instruction put
          false on the stack.",
    undefined ),

  new LearningStep(14, undefined, undefined),

  new LearningStep(15, undefined, undefined),

  new LearningStep(16, undefined, undefined),

  new LearningStep(17, undefined, undefined),

  new LearningStep(18, undefined, undefined),

  new LearningStep(19, undefined, undefined),

  new LearningStep(
    19,
    undefined,

    new Question(
      "What value is on the stack?",
      [true,10,-10],
      ["Secret is 8 and guess is 7. Are they equal?","The
            answer is not a number, just a boolean value.","The
             answer is not a number, just a boolean value."] )
            ),

  new LearningStep(20, undefined, undefined),

  new LearningStep(
    21,
    "Look at are_they_equal in the namespace. The expression \"
          secret == guess\" left false on the stack, so it was
          assigned to that variable.\n\n
    ",
    undefined ),

  new LearningStep(
    21,
    "Let's see how the equality operator works and check our
          understanding! This operator gets used a lot in
          programs.\n\n
    ",
    undefined ),

  new LearningStep(21, undefined, undefined),

  new LearningStep(22, undefined, undefined),

  new LearningStep(23, undefined, undefined),

  new LearningStep(24, undefined, undefined),

  new LearningStep(
    24,
    undefined,

    new Question(
      "What value is on the stack?",
      [true,[false],[true]],
      ["What were the two values? Are they equal?","The answer
            is not an array, just a boolean value.","The answer
             is not an array, just a boolean value."] ) ),

  new LearningStep(25, undefined, undefined),

  new LearningStep(26, undefined, undefined),

  new LearningStep(27, undefined, undefined),

  new LearningStep(28, undefined, undefined),

  new LearningStep(
    28,
    undefined,

    new Question(
```

```
    "What value is on the stack?",
    [false,[false],[true]],
    ["The whole value on each side is used.","The answer is
          not an array, just a boolean value.","The answer is
           not an array, just a boolean value."] ) ),

new LearningStep(29, undefined, undefined),

new LearningStep(30, undefined, undefined),

new LearningStep(31, undefined, undefined),

new LearningStep(32, undefined, undefined),

new LearningStep(
  32,
  undefined,

  new Question(
    "What value is on the stack?",
    [false,[true],[false]],
    ["","The answer is not an array, just a boolean value.","
          The answer is not an array, just a boolean value."]
          ) ),

new LearningStep(33, undefined, undefined),

new LearningStep(34, undefined, undefined),

new LearningStep(35, undefined, undefined),

new LearningStep(36, undefined, undefined),

new LearningStep(
  36,
  undefined,

  new Question(
    "What value is on the stack?",
    [true,[false],[true]],
    ["","The answer is not an array, just a boolean value.","
          The answer is not an array, just a boolean value."]
          ) ),

new LearningStep(37, undefined, undefined),

new LearningStep(38, undefined, undefined),

new LearningStep(39, undefined, undefined),

new LearningStep(40, undefined, undefined),

new LearningStep(41, undefined, undefined),

new LearningStep(
  41,
  undefined,

  new Question(
    "What value is on the stack?",
    [true,[true],[false]],
    ["Is negative one equal to one? Each side of the == gets
          fully evaluated then compared.","The answer is not
          an array, just a boolean value.","The answer is not
           an array, just a boolean value."] ) ),

new LearningStep(42, undefined, undefined),

new LearningStep(43, undefined, undefined),

new LearningStep(44, undefined, undefined),

new LearningStep(45, undefined, undefined),

new LearningStep(
  45,
  undefined,

  new Question(
    "What value is on the stack?",
    [false,[true],[false]],
    ["","The answer is not an array, just a boolean value.","
          The answer is not an array, just a boolean value."]
          ) ),

new LearningStep(
  46,
  "You can also use == on true and false values",
  undefined ),

new LearningStep(46, undefined, undefined),

new LearningStep(47, undefined, undefined),

new LearningStep(48, undefined, undefined),

new LearningStep(49, undefined, undefined),

new LearningStep(
  49,
  undefined,

  new Question(
    "What value is on the stack?",
    [false,[true],[false]],
    ["","The answer is not an array, just a boolean value.","
          The answer is not an array, just a boolean value."]
          ) ),

new LearningStep(50, undefined, undefined),

new LearningStep(51, undefined, undefined),

new LearningStep(52, undefined, undefined),

new LearningStep(53, undefined, undefined),

new LearningStep(
  53,
  undefined,

  new Question(
    "What value is on the stack?",
    [false,[false],[true]],
    ["False gets compared to false. Are they equal?","The
          answer is not an array, just a boolean value.","The
           answer is not an array, just a boolean value."] )
          ),

new LearningStep(
  54,
  "These will put false on the stack",
  undefined ),

new LearningStep(54, undefined, undefined),

new LearningStep(55, undefined, undefined),

new LearningStep(56, undefined, undefined),

new LearningStep(57, undefined, undefined),

new LearningStep(
  57,
  undefined,

  new Question(
    "What value is on the stack?",
    [true,[true],[false]],
    ["","The answer is not an array, just a boolean value.","
          The answer is not an array, just a boolean value."]
          ) ),

new LearningStep(58, undefined, undefined),

new LearningStep(59, undefined, undefined),

new LearningStep(60, undefined, undefined),

new LearningStep(61, undefined, undefined),

new LearningStep(
  61,
  undefined,

  new Question(
    "What value is on the stack?",
    [true,[true],[false]],
    ["","The answer is not an array, just a boolean value.","
          The answer is not an array, just a boolean value."]
```

```
        ) ),

new LearningStep(
  62,
  "Parentheses make the computer\ndo some instructions first
       .\nThis can change the results of\nan expression.",
  undefined ),

new LearningStep(62, undefined, undefined),

new LearningStep(63, undefined, undefined),

new LearningStep(64, undefined, undefined),

new LearningStep(65, undefined, undefined),

new LearningStep(66, undefined, undefined),

new LearningStep(67, undefined, undefined),

new LearningStep(
  67,
  undefined,

  new Question(
    "What value is on the stack?",
    [false,[true],[false]],
    ["False == False gets done first because it is inside
         parentheses. Expressions inside parentheses get
         done first.","The answer is not an array, just a
         boolean value.","The answer is not an array, just a
          boolean value."] ) ),

new LearningStep(68, undefined, undefined),

new LearningStep(69, undefined, undefined),

new LearningStep(70, undefined, undefined),

new LearningStep(71, undefined, undefined),

new LearningStep(72, undefined, undefined),

new LearningStep(73, undefined, undefined),

new LearningStep(
  73,
  undefined,

  new Question(
    "What value is on the stack?",
    [false,[true],[false]],
    ["The expression inside the parentheses is evaluated
         first. Go check what it was by stepping back.","The
          answer is not an array, just a boolean value.","
         The answer is not an array, just a boolean value."]
          ) ),

new LearningStep(74, undefined, undefined),

new LearningStep(75, undefined, undefined),

new LearningStep(76, undefined, undefined),

new LearningStep(77, undefined, undefined),

new LearningStep(78, undefined, undefined),

new LearningStep(79, undefined, undefined),

new LearningStep(
  79,
  undefined,

  new Question(
    "What value is on the stack?",
    [true,[false],[true]],
    ["","The answer is not an array, just a boolean value.","
         The answer is not an array, just a boolean value."]
          ) ),

new LearningStep(
  80,
  "There's another operator for not equal to, written as !=\n
       \n
  You can read the exclamation point as \"not\". For example,
        you can read 1 != 2 as \"1 is not equal to 2?\" and
       it will leave *true* on the stack. ",
  undefined ),

new LearningStep(80, undefined, undefined),

new LearningStep(81, undefined, undefined),

new LearningStep(82, undefined, undefined),

new LearningStep(83, undefined, undefined),

new LearningStep(
  83,
  undefined,

  new Question(
    "What value is on the stack?",
    [false,[true],[false]],
    ["Is false not equal to true?","The answer is not an
         array, just a boolean value.","The answer is not an
          array, just a boolean value."] ) ),

new LearningStep(84, undefined, undefined),

new LearningStep(85, undefined, undefined),

new LearningStep(86, undefined, undefined),

new LearningStep(87, undefined, undefined),

new LearningStep(
  87,
  undefined,

  new Question(
    "What value is on the stack?",
    [false,[false],[true]],
    ["","The answer is not an array, just a boolean value.","
         The answer is not an array, just a boolean value."]
          ) ),

new LearningStep(88, undefined, undefined),

new LearningStep(89, undefined, undefined),

new LearningStep(90, undefined, undefined),

new LearningStep(91, undefined, undefined),

new LearningStep(
  91,
  undefined,

  new Question(
    "What value is on the stack?",
    [true,[true],[false]],
    ["","The answer is not an array, just a boolean value.","
         The answer is not an array, just a boolean value."]
          ) ),

new LearningStep(92, undefined, undefined),

new LearningStep(93, undefined, undefined),

new LearningStep(94, undefined, undefined),

new LearningStep(95, undefined, undefined),

new LearningStep(
  95,
  undefined,

  new Question(
    "What value is on the stack?",
    [false,[false],[true]],
    ["","The answer is not an array, just a boolean value.","
         The answer is not an array, just a boolean value."]
          ) )]
                  ) /* end LearningSteps */
                  ) /* end Action */,
new Action (
                  new Program(`
!true;
```

```
!false;
! true;
! false;

var a = false;
var b = true;

a == b;
!a == b;
a == !b;

!(a == b);
```

```
            `
            ) /* end Program */,
                    new LearningSteps(
        /* Learning Steps */
        [
  new LearningStep(
    0,
    "## The Not Operator\n\n
    There is a unary operator for boolean values called the **
        not** operator. It is written with\nan **exclamation
        point!**\n\n
    It take the top value on the stack and gives the opposite.\
        n\n
    So if *true* is on the stack,\nit replaces it with *false
        *.\nIf *false* is on the stack,\nit replaces it with
        *true*.\n\n
    Watch and see how the operator works.",
    undefined ),

  new LearningStep(0, undefined, undefined),

  new LearningStep(1, undefined, undefined),

  new LearningStep(2, undefined, undefined),

  new LearningStep(3, undefined, undefined),

  new LearningStep(4, undefined, undefined),

  new LearningStep(5, undefined, undefined),

  new LearningStep(6, undefined, undefined),

  new LearningStep(7, undefined, undefined),

  new LearningStep(8, undefined, undefined),

  new LearningStep(
    8,
    undefined,

    new Question(
      "What does !true leave on the stack?",
      [true,10,-10],
      ["","The answer is a boolean, not a number.","The answer
          is a boolean, not a number."] ) ),

  new LearningStep(9, undefined, undefined),

  new LearningStep(10, undefined, undefined),

  new LearningStep(11, undefined, undefined),

  new LearningStep(
    11,
    undefined,

    new Question(
      "What does !false leave on the stack?",
      [false,10,-10],
      ["","The answer is a boolean, not a number.","The answer
          is a boolean, not a number."] ) ),

  new LearningStep(
    12,
    "You can use operators with variables, since they all use
        the stack.",
    undefined ),

  new LearningStep(13, undefined, undefined),

  new LearningStep(14, undefined, undefined),

  new LearningStep(15, undefined, undefined),

  new LearningStep(16, undefined, undefined),

  new LearningStep(17, undefined, undefined),

  new LearningStep(18, undefined, undefined),

  new LearningStep(19, undefined, undefined),

  new LearningStep(20, undefined, undefined),

  new LearningStep(21, undefined, undefined),

  new LearningStep(22, undefined, undefined),

  new LearningStep(23, undefined, undefined),

  new LearningStep(24, undefined, undefined),

  new LearningStep(25, undefined, undefined),

  new LearningStep(
    25,
    undefined,

    new Question(
      "What is on the stack?",
      [0,1,true],
      [] ) ),

  new LearningStep(26, undefined, undefined),

  new LearningStep(27, undefined, undefined),

  new LearningStep(
    27,
    undefined,

    new Question(
      "What is on the stack after this instruction executes?",
      [false,[false],[true]],
      ["The ! takes a, which is false, and negates it. What is
          not false?","The answer is not an array, just a
          boolean value.","The answer is not an array, just a
           boolean value."] ) ),

  new LearningStep(28, undefined, undefined),

  new LearningStep(29, undefined, undefined),

  new LearningStep(30, undefined, undefined),

  new LearningStep(
    30,
    undefined,

    new Question(
      "What is on the stack?",
      [0,1,false],
      [] ) ),

  new LearningStep(31, undefined, undefined),

  new LearningStep(32, undefined, undefined),

  new LearningStep(33, undefined, undefined),

  new LearningStep(34, undefined, undefined),

  new LearningStep(35, undefined, undefined),

  new LearningStep(
    35,
    undefined,

    new Question(
      "What is on the stack?",
      [0,1,false],
      [] ) ),

  new LearningStep(36, undefined, undefined),
```

```
  new LearningStep(37, undefined, undefined),

  new LearningStep(38, undefined, undefined),

  new LearningStep(39, undefined, undefined),

  new LearningStep(40, undefined, undefined),

  new LearningStep(
    40,
    undefined,

    new Question(
      "What is on the stack?",
      [0,1,false],
      ["","","false == true gets done first, pushing false.
          Then the ! changes the value on the stack to its
          opposite."] ) ),

  new LearningStep(
    40,
    " **There's a lot here and you're putting in the work
        needed to learn it!**\n\n
    Thank you for sticking with us. It takes a while to learn
        all these simple rules, and they can be put together
        to do incredible things!\n",
    undefined )]
                    ) /* end LearningSteps */
                    ) /* end Action */,
new Action (
                    new Program(`


var x_1 = 1 > 2;
var x_2 = 2 > 1;

var x_3 = 1 >= 1;
var x_4 = 1 > 1;
var x_5 = 1 >= 0;
var x_6 = 1 > 0;

var x_7 = 1 <= 0;
var x_8 = 1 < 0;
var x_9 = 0 <= 1;
var x_10 = 0 < 1;
var x_11 = 0 <= 0;
var x_12 = 0 < 0;




            `
            ) /* end Program */,
                    new LearningSteps(
        /* Learning Steps */
        [
  new LearningStep(
    0,
    "## Comparison Operators\n\n
    Here are all the comparison operators and their English
        meanings:\n1) \\>        greater than\n2) \\<
        less than\n3) \\>=         greater than or equal to\n4
        ) \\<=         less than or equal to\n5) ==
        equal to\n6) !=         not equal to\n\n
    They evaluate to **boolean** values. Boolean values work
        like other values: they can go on the stack or be
        stored in variables.\n\n
    \n\n
    ",
    undefined ),

  new LearningStep(
    0,
    "As you step through these expressions, **read the
        expression mentally as a sentence** before answering
        it. For example, the first one is   is   one greater
        than 2?   \n",
    undefined ),

  new LearningStep(0, undefined, undefined),

  new LearningStep(1, undefined, undefined),

  new LearningStep(2, undefined, undefined),

  new LearningStep(3, undefined, undefined),

  new LearningStep(4, undefined, undefined),

  new LearningStep(5, undefined, undefined),

  new LearningStep(
    5,
    undefined,

    new Question(
      "What value is on the stack?",
      [true,[true],[false]],
      ["","The answer is not an array, just a boolean value.","
          The answer is not an array, just a boolean value."]
          ) ),

  new LearningStep(6, undefined, undefined),

  new LearningStep(7, undefined, undefined),

  new LearningStep(8, undefined, undefined),

  new LearningStep(9, undefined, undefined),

  new LearningStep(10, undefined, undefined),

  new LearningStep(11, undefined, undefined),

  new LearningStep(12, undefined, undefined),

  new LearningStep(
    12,
    undefined,

    new Question(
      "What value is on the stack?",
      [false,[false],[true]],
      ["","The answer is not an array, just a boolean value.","
          The answer is not an array, just a boolean value."]
          ) ),

  new LearningStep(13, undefined, undefined),

  new LearningStep(14, undefined, undefined),

  new LearningStep(15, undefined, undefined),

  new LearningStep(16, undefined, undefined),

  new LearningStep(17, undefined, undefined),

  new LearningStep(18, undefined, undefined),

  new LearningStep(19, undefined, undefined),

  new LearningStep(
    19,
    undefined,

    new Question(
      "What value is on the stack?",
      [false,[false],[true]],
      ["","The answer is not an array, just a boolean value.","
          The answer is not an array, just a boolean value."]
          ) ),

  new LearningStep(20, undefined, undefined),

  new LearningStep(21, undefined, undefined),

  new LearningStep(22, undefined, undefined),

  new LearningStep(23, undefined, undefined),

  new LearningStep(24, undefined, undefined),

  new LearningStep(25, undefined, undefined),

  new LearningStep(26, undefined, undefined),

  new LearningStep(
    26,
```

```
  undefined,

  new Question(
    "What value is on the stack?",
    [true,[false],[true]],
    ["","The answer is not an array, just a boolean value.","
          The answer is not an array, just a boolean value."]
          )),

new LearningStep(27, undefined, undefined),

new LearningStep(28, undefined, undefined),

new LearningStep(29, undefined, undefined),

new LearningStep(30, undefined, undefined),

new LearningStep(31, undefined, undefined),

new LearningStep(32, undefined, undefined),

new LearningStep(33, undefined, undefined),

new LearningStep(
  33,
  undefined,

  new Question(
    "What value is on the stack?",
    [false,[true],[false]],
    ["","The answer is not an array, just a boolean value.","
          The answer is not an array, just a boolean value."]
          )),

new LearningStep(34, undefined, undefined),

new LearningStep(35, undefined, undefined),

new LearningStep(36, undefined, undefined),

new LearningStep(37, undefined, undefined),

new LearningStep(38, undefined, undefined),

new LearningStep(39, undefined, undefined),

new LearningStep(40, undefined, undefined),

new LearningStep(
  40,
  undefined,

  new Question(
    "What value is on the stack?",
    [false,[false],[true]],
    ["","The answer is not an array, just a boolean value.","
          The answer is not an array, just a boolean value."]
          )),

new LearningStep(41, undefined, undefined),

new LearningStep(42, undefined, undefined),

new LearningStep(43, undefined, undefined),

new LearningStep(44, undefined, undefined),

new LearningStep(45, undefined, undefined),

new LearningStep(46, undefined, undefined),

new LearningStep(47, undefined, undefined),

new LearningStep(
  47,
  undefined,

  new Question(
    "What value is on the stack?",
    [true,[true],[false]],
    ["","The answer is not an array, just a boolean value.","
          The answer is not an array, just a boolean value."]
          )),

new LearningStep(48, undefined, undefined),

new LearningStep(49, undefined, undefined),

new LearningStep(50, undefined, undefined),

new LearningStep(51, undefined, undefined),

new LearningStep(52, undefined, undefined),

new LearningStep(53, undefined, undefined),

new LearningStep(54, undefined, undefined),

new LearningStep(
  54,
  undefined,

  new Question(
    "What value is on the stack?",
    [true,[true],[false]],
    ["","The answer is not an array, just a boolean value.","
          The answer is not an array, just a boolean value."]
          )),

new LearningStep(55, undefined, undefined),

new LearningStep(56, undefined, undefined),

new LearningStep(57, undefined, undefined),

new LearningStep(58, undefined, undefined),

new LearningStep(59, undefined, undefined),

new LearningStep(60, undefined, undefined),

new LearningStep(61, undefined, undefined),

new LearningStep(
  61,
  undefined,

  new Question(
    "What value is on the stack?",
    [false,[false],[true]],
    ["","The answer is not an array, just a boolean value.","
          The answer is not an array, just a boolean value."]
          )),

new LearningStep(62, undefined, undefined),

new LearningStep(63, undefined, undefined),

new LearningStep(64, undefined, undefined),

new LearningStep(65, undefined, undefined),

new LearningStep(66, undefined, undefined),

new LearningStep(67, undefined, undefined),

new LearningStep(68, undefined, undefined),

new LearningStep(
  68,
  undefined,

  new Question(
    "What value is on the stack?",
    [false,[true],[false]],
    ["","The answer is not an array, just a boolean value.","
          The answer is not an array, just a boolean value."]
          )),

new LearningStep(69, undefined, undefined),

new LearningStep(70, undefined, undefined),

new LearningStep(71, undefined, undefined),

new LearningStep(72, undefined, undefined),

new LearningStep(73, undefined, undefined),

new LearningStep(74, undefined, undefined),
```

```
  new LearningStep(75, undefined, undefined),

  new LearningStep(
    75,
    undefined,

    new Question(
      "What value is on the stack?",
      [false,[false],[true]],
      ["","The answer is not an array, just a boolean value.","
           The answer is not an array, just a boolean value."]
       )),

  new LearningStep(76, undefined, undefined),

  new LearningStep(77, undefined, undefined),

  new LearningStep(78, undefined, undefined),

  new LearningStep(79, undefined, undefined),

  new LearningStep(80, undefined, undefined),

  new LearningStep(81, undefined, undefined),

  new LearningStep(82, undefined, undefined),

  new LearningStep(
    82,
    undefined,

    new Question(
      "What value is on the stack?",
      [true,[false],[true]],
      ["","The answer is not an array, just a boolean value.","
           The answer is not an array, just a boolean value."]
       )),

  new LearningStep(
    83,
    " ",
    undefined ),

  new LearningStep(
    83,
    "Awesome! Onward!",
    undefined )]
                    ) /* end LearningSteps */
                    ) /* end Action */,
new Action (
                    new Program(`


true && true;
true && false;
false && true;
false && false;

true || true;
true || false;
false || true;
false || false;


true && (true || false);
false && (false || true);
true || (false && true);

            `
            ) /* end Program */,
                    new LearningSteps(
        /* Learning Steps */
        [
  new LearningStep(
    0,
    "## \"Or\" and \"And\" Operators\n\n
    Here are two more boolean operators that are very\
         nimportant, but the language designers chose to use\
         nsome strange symbols to represent them.\n\n
    The **And** Operator\n\n
    (left) **&&** (right)\n\n
    This operator leaves true on the stack if both left and
         right are true.\nOtherwise it leaves false.",
    undefined ),

new LearningStep(
  0,
  "The **And** Operator\n\n
  (left) **&&** (right)\n\n
  This operator leaves true on the stack if both left and
       right are true.\nOtherwise it leaves false.\n\n
  Let's try some questions!",
  undefined ),

new LearningStep(0, undefined, undefined),

new LearningStep(1, undefined, undefined),

new LearningStep(2, undefined, undefined),

new LearningStep(3, undefined, undefined),

new LearningStep(4, undefined, undefined),

new LearningStep(5, undefined, undefined),

new LearningStep(6, undefined, undefined),

new LearningStep(7, undefined, undefined),

new LearningStep(8, undefined, undefined),

new LearningStep(9, undefined, undefined),

new LearningStep(10, undefined, undefined),

new LearningStep(11, undefined, undefined),

new LearningStep(12, undefined, undefined),

new LearningStep(13, undefined, undefined),

new LearningStep(14, undefined, undefined),

new LearningStep(15, undefined, undefined),

new LearningStep(
  15,
  undefined,

  new Question(
    "What is on the stack?",
    [0,1,true],
    [] ) ),

new LearningStep(
  16,
  "The **Or** operator\n\n
  (left) || (right)\n\n
  This operator leaves true on the stack if at least one of
       left or right is true. Otherwise it leaves false.",
  undefined ),

new LearningStep(16, undefined, undefined),

new LearningStep(17, undefined, undefined),

new LearningStep(18, undefined, undefined),

new LearningStep(19, undefined, undefined),

new LearningStep(20, undefined, undefined),

new LearningStep(21, undefined, undefined),

new LearningStep(22, undefined, undefined),

new LearningStep(23, undefined, undefined),

new LearningStep(24, undefined, undefined),

new LearningStep(25, undefined, undefined),

new LearningStep(26, undefined, undefined),

new LearningStep(27, undefined, undefined),

new LearningStep(
  27,
```

```
  undefined,

  new Question(
    "What is on the stack?",
    [0,1,false],
    [] ) ),

new LearningStep(28, undefined, undefined),

new LearningStep(29, undefined, undefined),

new LearningStep(30, undefined, undefined),

new LearningStep(
  31,
  " ",
  undefined ),

new LearningStep(
  31,
  "Let's try some questions and review parentheses. *Read the
       expressions mentally in English* before answering
      them.",
  undefined ),

new LearningStep(32, undefined, undefined),

new LearningStep(33, undefined, undefined),

new LearningStep(34, undefined, undefined),

new LearningStep(35, undefined, undefined),

new LearningStep(36, undefined, undefined),

new LearningStep(37, undefined, undefined),

new LearningStep(
  37,
  undefined,

  new Question(
    "What is on the stack?",
    [0,1,false],
    [] ) ),

new LearningStep(38, undefined, undefined),

new LearningStep(39, undefined, undefined),

new LearningStep(40, undefined, undefined),

new LearningStep(41, undefined, undefined),

new LearningStep(42, undefined, undefined),

new LearningStep(43, undefined, undefined),

new LearningStep(
  43,
  undefined,

  new Question(
    "What is on the stack?",
    [0,1,true],
    [] ) ),

new LearningStep(44, undefined, undefined),

new LearningStep(45, undefined, undefined),

new LearningStep(46, undefined, undefined),

new LearningStep(47, undefined, undefined),

new LearningStep(48, undefined, undefined),

new LearningStep(49, undefined, undefined),

new LearningStep(
  49,
  undefined,

  new Question(
    "What is on the stack?",
    [0,1,false],
    [] ) )
                    ) /* end LearningSteps */
                    ) /* end Action */]),
new KnowledgeUnit("If statements",
          "
If statements allow computers to do some set
of instructions if a condition is true or not.

The condition is inside of the parentheses () that
follow the if statement. The brackets {}
tell the computer which parts of the program
to do or not do. If there were no brackets,
it would not know where to go.",
           [
new Action (
                    new Program(`


var x = 0;
if ( 10 > 0 ){
    /* the computer will execute inside here
       because the condition is true
       and that leaves true on the stack

       we put x = 100000; here
       just so you can see some code
       execute inside the if */
    x = 1000000;
}

x;
var x = 0;
if ( 0 > 10 ){
    /* the computer will NOT execute inside here
       because the condition is false
       and that leaves false on the stack */
    x = 1000000;
}
x;



var x = 0;
if ( 10 != 0 ){
    /* the computer will execute inside here
       because 10 is not eqaul to 0
       and that leaves true on the stack */

    x = 1000000;
}
x;

var x = 0;
if ( 0 == 10 ){
    /* the computer will NOT execute inside here
       because 0 is equal to 10
       leaves false on the stack */
    x = 1000000;
}

var x = 0;
if ( 0 != 10 ){
    /* the computer will execute inside here
       because 0 is not equal to 10
       leaves false on the stack */
    x = 1000000;
}
x;


x = 0;
if ( true ){
    /* the computer will execute inside here
       because the condition is true
       and that leaves true on the stack */
    x = 1000000;
}
x;

x = 0;
if (false){
    /* the computer will NOT execute inside here
       because the condition is false
       and that leaves false on the stack */
```

```
    x = 1000000;
}
x;

var y = true;
x = 0;
if ( y ){
    /* the computer looks up y
       pushes true onto the stack
       then, because true is on the stack
       it does this code in here */
    x = 70;
}
x;

var z = false;
x = 0;
if ( z ){
    /* the computer looks up z
       pushes false onto the stack
       then, because false is on the stack
       it SKIPS OVER this code
       and continues after the } */
    // the computer won't execute in here
    x = -33;
}

x;

x = 0;
if ( true == true ){
    // the computer will execute inside here
    x = 70;
}

x;

x = 1;
if ( true == false ){
    // the computer won't execute in here
    x = -33;
}
x;

x = 1;
if ( ! (true) ){
    x = 20;
}
x;

x = 1;
if ( ! (false)){
    x = 45;
}
x;
```

```
            `
            ) /* end Program */,
                    new LearningSteps(
        /* Learning Steps */
        [
  new LearningStep(
    0,
    "## If statements\n\n
    Now it's time to use what you learned about boolean values
         and operators!\n\n
    Before this, the computer would execute all instructions
         created from the code.\n\n
    If statements allow computers to do some set of
         instructions if a **condition** is true or not. \n\n
    They look like this\n```\nif ( condition )\n{\n   code goes
         inside the { }'s\n}\n```\n\n
    Let's step through one to see how it works.",
    undefined ),

  new LearningStep(0, undefined, undefined),

  new LearningStep(1, undefined, undefined),

  new LearningStep(2, undefined, undefined),

  new LearningStep(3, undefined, undefined),

  new LearningStep(4, undefined, undefined),

  new LearningStep(
    4,
    "Carefully step through the code and read all the
         explanations for the instructions, and look at the
         stack.",
    undefined ),

  new LearningStep(5, undefined, undefined),

  new LearningStep(
    5,
    "The **condition** is what we call\nthe expression inside
         of the parentheses that\nfollow the if.\n\n
    Here the condition is 10 > 0\n\n
    If the condition leaves true on the stack, the computer
         executes the code within the **curly braces** { }.\n\
         n
    Read the instruction descriptions and step through this
         carefully.",
    undefined ),

  new LearningStep(6, undefined, undefined),

  new LearningStep(7, undefined, undefined),

  new LearningStep(8, undefined, undefined),

  new LearningStep(
    9,
    "Now, 10 > 0 left true on the stack, as you can see. This
         means the condition evaluated to true. Now the code
         inside the { } will be executed.",
    undefined ),

  new LearningStep(10, undefined, undefined),

  new LearningStep(11, undefined, undefined),

  new LearningStep(12, undefined, undefined),

  new LearningStep(13, undefined, undefined),

  new LearningStep(
    14,
    "x has a new value now because the condition 10 > 0 was
         true. You can see it on the stack.",
    undefined ),

  new LearningStep(
    14,
    "x has a new value now because the condition 10 > 0 was
         true. You can see it on the stack. Let's see what
         happens when the condition evaluates to false (it
         leaves false on the stack).",
    undefined ),

  new LearningStep(15, undefined, undefined),

  new LearningStep(16, undefined, undefined),

  new LearningStep(17, undefined, undefined),

  new LearningStep(18, undefined, undefined),

  new LearningStep(19, undefined, undefined),

  new LearningStep(20, undefined, undefined),

  new LearningStep(21, undefined, undefined),

  new LearningStep(22, undefined, undefined),

  new LearningStep(23, undefined, undefined),

  new LearningStep(
    24,
    "What did 0 > 10 evaluate to (leave on the stack)?",

    new Question(
      "What did 0 > 10 evaluate to (leave on the stack)?",
      [true,10,-10],
      ["0 > 10 is a boolean operator that leaves true if the
```

```
        left side is greater than the right, otherwise it
        leaves false.","The answer is a boolean, not a
        number.","The answer is a boolean, not a number."]
        ) ),

new LearningStep(
  24,
  "**Because the condition left false on the stack** (it was
       0 > 10), the computer **skips over the instructions**
       from code between the next { }. It does this by
       changing the next instruction to execute. Watch it
       skip over.",
  undefined ),

new LearningStep(
  25,
  "Woah, the computer just skipped over the code inside the {
       }. That's the first time you've seen that happen.\n\
       n
  Thus, the value in x didn't change. It is still 0.\n",
  undefined ),

new LearningStep(26, undefined, undefined),

new LearningStep(27, undefined, undefined),

new LearningStep(28, undefined, undefined),

new LearningStep(29, undefined, undefined),

new LearningStep(30, undefined, undefined),

new LearningStep(31, undefined, undefined),

new LearningStep(
  31,
  "Let's watch a few more. \n",
  undefined ),

new LearningStep(32, undefined, undefined),

new LearningStep(33, undefined, undefined),

new LearningStep(34, undefined, undefined),

new LearningStep(35, undefined, undefined),

new LearningStep(36, undefined, undefined),

new LearningStep(
  36,
  "Where do you think the computer will execute next?",
  undefined ),

new LearningStep(37, undefined, undefined),

new LearningStep(38, undefined, undefined),

new LearningStep(39, undefined, undefined),

new LearningStep(40, undefined, undefined),

new LearningStep(41, undefined, undefined),

new LearningStep(42, undefined, undefined),

new LearningStep(43, undefined, undefined),

new LearningStep(44, undefined, undefined),

new LearningStep(45, undefined, undefined),

new LearningStep(46, undefined, undefined),

new LearningStep(47, undefined, undefined),

new LearningStep(48, undefined, undefined),

new LearningStep(49, undefined, undefined),

new LearningStep(50, undefined, undefined),

new LearningStep(
  50,
  "Does 0 equal 10? Will the code in this if statement's { }
       get executed?",
  undefined ),

new LearningStep(51, undefined, undefined),

new LearningStep(
  52,
  "The code between the last if statement's { } got skipped
       over. We also call this *jumping* over code.",
  undefined ),

new LearningStep(53, undefined, undefined),

new LearningStep(54, undefined, undefined),

new LearningStep(55, undefined, undefined),

new LearningStep(56, undefined, undefined),

new LearningStep(57, undefined, undefined),

new LearningStep(58, undefined, undefined),

new LearningStep(59, undefined, undefined),

new LearningStep(60, undefined, undefined),

new LearningStep(61, undefined, undefined),

new LearningStep(62, undefined, undefined),

new LearningStep(63, undefined, undefined),

new LearningStep(64, undefined, undefined),

new LearningStep(65, undefined, undefined),

new LearningStep(66, undefined, undefined),

new LearningStep(
  66,
  undefined,

  new Question(
    "What value is on the stack?",
    [0,5,10],
    ["0 != 10 leaves true on the stack, so the if body
         executes. It was 0 before the if statement though!"
         ,"","10 is inside the condition and x doesn't get
         set to 10. The condition only puts something on the
          stack."] ) ),

new LearningStep(
  67,
  "These next if statements might look odd at first.\n\n
  true just puts true on the stack. Thus, the condition in
       the next if statement will evaluate to true. \n\n
  We're showing you **the computer just uses what is on the
       stack** to decide to do the code in the { } or not.",
  undefined ),

new LearningStep(67, undefined, undefined),

new LearningStep(68, undefined, undefined),

new LearningStep(69, undefined, undefined),

new LearningStep(70, undefined, undefined),

new LearningStep(71, undefined, undefined),

new LearningStep(72, undefined, undefined),

new LearningStep(73, undefined, undefined),

new LearningStep(74, undefined, undefined),

new LearningStep(75, undefined, undefined),

new LearningStep(
  75,
  undefined,

  new Question(
    "What value is on the stack?",
```

```
    [0,100,5],
    ["The code inside an if statement gets executed as normal
        . Here x is being set to what?","",""] ) ),

new LearningStep(76, undefined, undefined),

new LearningStep(77, undefined, undefined),

new LearningStep(78, undefined, undefined),

new LearningStep(79, undefined, undefined),

new LearningStep(80, undefined, undefined),

new LearningStep(81, undefined, undefined),

new LearningStep(
  81,
  "Let's do some more questions to check your understanding."
      ,
  undefined ),

new LearningStep(82, undefined, undefined),

new LearningStep(83, undefined, undefined),

new LearningStep(84, undefined, undefined),

new LearningStep(85, undefined, undefined),

new LearningStep(
  85,
  undefined,

  new Question(
    "What value is on the stack?",
    [10,1000000,1],
    ["","The condition for the if statement left false on the
         stack. The if condition is defined to skip over
        the code in the { } when that happens..","",""] ) ),

new LearningStep(86, undefined, undefined),

new LearningStep(87, undefined, undefined),

new LearningStep(88, undefined, undefined),

new LearningStep(89, undefined, undefined),

new LearningStep(90, undefined, undefined),

new LearningStep(91, undefined, undefined),

new LearningStep(92, undefined, undefined),

new LearningStep(93, undefined, undefined),

new LearningStep(94, undefined, undefined),

new LearningStep(95, undefined, undefined),

new LearningStep(
  95,
  undefined,

  new Question(
    "What value is on the stack after this instruction?",
    [false,[true],[false]],
    ["Look a few lines up to see y's definition.","The answer
         is not an array, just a boolean value.","The
        answer is not an array, just a boolean value."] ) )
        ,

new LearningStep(96, undefined, undefined),

new LearningStep(97, undefined, undefined),

new LearningStep(98, undefined, undefined),

new LearningStep(99, undefined, undefined),

new LearningStep(100, undefined, undefined),

new LearningStep(101, undefined, undefined),

new LearningStep(
  101,
  undefined,

  new Question(
    "What value is on the stack?",
    [10,0,5],
    ["","The condition left true on the stack, so the code
         inside the { } did execute.","",""] ) ),

new LearningStep(102, undefined, undefined),

new LearningStep(103, undefined, undefined),

new LearningStep(104, undefined, undefined),

new LearningStep(105, undefined, undefined),

new LearningStep(106, undefined, undefined),

new LearningStep(107, undefined, undefined),

new LearningStep(108, undefined, undefined),

new LearningStep(109, undefined, undefined),

new LearningStep(110, undefined, undefined),

new LearningStep(111, undefined, undefined),

new LearningStep(112, undefined, undefined),

new LearningStep(113, undefined, undefined),

new LearningStep(114, undefined, undefined),

new LearningStep(
  114,
  undefined,

  new Question(
    "What value is on the stack?",
    [10,1,5],
    ["","","",""] ) ),

new LearningStep(115, undefined, undefined),

new LearningStep(116, undefined, undefined),

new LearningStep(117, undefined, undefined),

new LearningStep(118, undefined, undefined),

new LearningStep(119, undefined, undefined),

new LearningStep(120, undefined, undefined),

new LearningStep(121, undefined, undefined),

new LearningStep(122, undefined, undefined),

new LearningStep(123, undefined, undefined),

new LearningStep(124, undefined, undefined),

new LearningStep(125, undefined, undefined),

new LearningStep(126, undefined, undefined),

new LearningStep(127, undefined, undefined),

new LearningStep(
  127,
  undefined,

  new Question(
    "What did the condition of this if statement evaluate to?
        ",
    [100,0,10],
    ["","Is true == true? If the expression between the ( )
         after the if is true, the code inside the { } gets
         executed.","",""] ) ),

new LearningStep(128, undefined, undefined),
```

```
new LearningStep(129, undefined, undefined),

new LearningStep(130, undefined, undefined),

new LearningStep(131, undefined, undefined),

new LearningStep(132, undefined, undefined),

new LearningStep(133, undefined, undefined),

new LearningStep(134, undefined, undefined),

new LearningStep(135, undefined, undefined),

new LearningStep(136, undefined, undefined),

new LearningStep(137, undefined, undefined),

new LearningStep(
  137,
  undefined,

  new Question(
    "What value is on the stack?",
    [10,-33,5],
    ["","The whole expression in between the ( ) gets
         evaluated then it's final value on the stack is
         used as the condition.",""] ) ),

new LearningStep(138, undefined, undefined),

new LearningStep(139, undefined, undefined),

new LearningStep(140, undefined, undefined),

new LearningStep(141, undefined, undefined),

new LearningStep(142, undefined, undefined),

new LearningStep(143, undefined, undefined),

new LearningStep(144, undefined, undefined),

new LearningStep(145, undefined, undefined),

new LearningStep(146, undefined, undefined),

new LearningStep(
  146,
  undefined,

  new Question(
    "What value is on the stack?",
    [5,0,20],
    ["","","The if did not execute. The entire expression
         between the ( ) gets evaluated, then the last value
          on the stack is used to determine to do the code
         inside or not."] ) ),

new LearningStep(147, undefined, undefined),

new LearningStep(148, undefined, undefined),

new LearningStep(149, undefined, undefined),

new LearningStep(150, undefined, undefined),

new LearningStep(151, undefined, undefined),

new LearningStep(152, undefined, undefined),

new LearningStep(153, undefined, undefined),

new LearningStep(154, undefined, undefined),

new LearningStep(155, undefined, undefined),

new LearningStep(156, undefined, undefined),

new LearningStep(157, undefined, undefined),

new LearningStep(158, undefined, undefined),

new LearningStep(
  158,
  undefined,

  new Question(
    "What value is on the stack?",
    [100,1,0],
    ["","What does not false evaluate to?",""] ) ),

 new LearningStep(
   158,
   "Phew, great job! Onwards!",
   undefined )]
                    ) /* end LearningSteps */
                    ) /* end Action */,
new Action (
                    new Program(`
var priceOfBag = 100;
var myMoney = 90;
var moneyINeed = 0;
var numberOfBagsIOwn = 0;

if (myMoney >= priceOfBag){
    myMoney = myMoney - priceOfBag;
    numberOfBagsIOwn = 1;
}

if ( ! (myMoney >= priceOfBag) ){
    moneyINeed = priceOfBag - myMoney;
}

moneyINeed = 0;

if (myMoney >= priceOfBag){
    myMoney = moneyIHave - priceOfBag;
    numberOfBagsIOwn = 1;
} else {
    moneyINeed = priceOfBag - myMoney;
}


var x = 1;
if ( x > 100 ){
    x = 100;
} else {
    x = -1;
}

x;


var x = 1;
if ( x > 0){
    x = 100;
} else {
    x = -1;
}

x;



            `
            ) /* end Program */,
                    new LearningSteps(
          /* Learning Steps */
          [
  new LearningStep(
    0,
    "## Using If statements\n\n
    How might we use an if statement?\n\n
    Let's look at a program that helps someone plan their
         spending. Imagine we're just looking at a small part
         that helps with buying a bag. If you can't afford the
          bag, it helps you save the money to do so.",
    undefined ),

  new LearningStep(
    1,
    " ",
    undefined ),

  new LearningStep(
    1,
    "This variable name is written in a style called camel case
        .\n\n
    What is **camelCase**?\n\n
```

```
  Take a phrase like word1 word2 word3. Then capitalize all
       of the words and put them together - for example,
       Word1Word2Word3. You can also make the first letter
       lower case - for example, word1Word2Word3. ",
  undefined ),

new LearningStep(
  1,
  "Why **camelCase**?\n\n
  We can't have spaces in variable names. We could write _
       instead of a space, but this makes the word longer.
       In the Java and Javascript communities, people tend
       to write in the style of camelCase. It's weird but
       you can learn to read it.",
  undefined ),

new LearningStep(2, undefined, undefined),

new LearningStep(3, undefined, undefined),

new LearningStep(4, undefined, undefined),

new LearningStep(5, undefined, undefined),

new LearningStep(6, undefined, undefined),

new LearningStep(
  6,
  "This represents the money I have.",
  undefined ),

new LearningStep(7, undefined, undefined),

new LearningStep(8, undefined, undefined),

new LearningStep(9, undefined, undefined),

new LearningStep(10, undefined, undefined),

new LearningStep(11, undefined, undefined),

new LearningStep(12, undefined, undefined),

new LearningStep(13, undefined, undefined),

new LearningStep(14, undefined, undefined),

new LearningStep(15, undefined, undefined),

new LearningStep(16, undefined, undefined),

new LearningStep(17, undefined, undefined),

new LearningStep(18, undefined, undefined),

new LearningStep(19, undefined, undefined),

new LearningStep(20, undefined, undefined),

new LearningStep(
  20,
  "We'll describe what the program does using human language.
       ",
  undefined ),

new LearningStep(
  20,
  "If I can afford the bag, I model me buying the bag by
       subtracting the price from my money. I also now own a
        bag, and I keep track of the number of bags I own in
        the variable numberOfBagsIOwn.",
  undefined ),

new LearningStep(21, undefined, undefined),

new LearningStep(22, undefined, undefined),

new LearningStep(23, undefined, undefined),

new LearningStep(
  24,
  " ",

  new Question(
    "What did the condition evaluate to?",
    [true,10,-10],
    [">= means greater than or equal to.","The answer is a
         boolean, not a number.","The answer is a boolean,
         not a number."] ) ),

new LearningStep(24, undefined, undefined),

new LearningStep(
  25,
  "So, if I can't afford the bag, I calculate the money I
       need. I save that amount in the money I need variable
       . We name this variable moneyINeed in the camelCase
       style.",
  undefined ),

new LearningStep(
  25,
  "If I can't afford the bag, I calculate the money I need. I
        save that amount in the money I need variable. We
       name this variable moneyINeed in the camelCase style.
       ",
  undefined ),

new LearningStep(26, undefined, undefined),

new LearningStep(27, undefined, undefined),

new LearningStep(28, undefined, undefined),

new LearningStep(29, undefined, undefined),

new LearningStep(30, undefined, undefined),

new LearningStep(31, undefined, undefined),

new LearningStep(32, undefined, undefined),

new LearningStep(33, undefined, undefined),

new LearningStep(34, undefined, undefined),

new LearningStep(35, undefined, undefined),

new LearningStep(
  35,
  "Now we might have more code that makes suggestions based
       on the money I need. We won't go through that here.\n
       \n
  Instead, we'll use this example to motivate another
       language feature - the **else** clause.",
  undefined ),

new LearningStep(
  35,
  "## Else\n\n
  Sometimes you want to execute some code if a condition is
       true, otherwise execute a different set of code.\n\n
  Language designers added a feature for that. Instead of
       having to write another, nearly identical if
       statement, you can add an **else** section after the
        if s { }. It looks like \n```\nif (condition) {\n
        some code goes here\n}\nelse {\n  the code to
       execute if condition was false\n}\n```\n",
  undefined ),

new LearningStep(
  36,
  "This line is here to start over fresh.\n\n
  We can imagine replacing the **two** if statements above
       with the **single** if - else statements below this
       line.",
  undefined ),

new LearningStep(37, undefined, undefined),

new LearningStep(38, undefined, undefined),

new LearningStep(39, undefined, undefined),

new LearningStep(40, undefined, undefined),

new LearningStep(41, undefined, undefined),

new LearningStep(42, undefined, undefined),
```

```
new LearningStep(
  43,
  " ",
  undefined ),

new LearningStep(
  43,
  "Now, **because the condition was false**, the computer
       will execute the statements inside the else { }",
  undefined ),

new LearningStep(44, undefined, undefined),

new LearningStep(45, undefined, undefined),

new LearningStep(46, undefined, undefined),

new LearningStep(47, undefined, undefined),

new LearningStep(48, undefined, undefined),

new LearningStep(
  48,
  undefined,

  new Question(
    " What is the value of moneyINeed?",
    [100,0,5],
    ["","",""] ) ),

new LearningStep(
  49,
  "This next if - else statement will check x then set it to
       -1 because the condition was false.",
  undefined ),

new LearningStep(49, undefined, undefined),

new LearningStep(50, undefined, undefined),

new LearningStep(51, undefined, undefined),

new LearningStep(52, undefined, undefined),

new LearningStep(53, undefined, undefined),

new LearningStep(54, undefined, undefined),

new LearningStep(55, undefined, undefined),

new LearningStep(56, undefined, undefined),

new LearningStep(57, undefined, undefined),

new LearningStep(58, undefined, undefined),

new LearningStep(59, undefined, undefined),

new LearningStep(60, undefined, undefined),

new LearningStep(61, undefined, undefined),

new LearningStep(62, undefined, undefined),

new LearningStep(63, undefined, undefined),

new LearningStep(64, undefined, undefined),

new LearningStep(
  64,
  undefined,

  new Question(
    "What value is on the stack?",
    [5,0,100],
    ["","","x > 100 put false on the stack. This causes the
         statements after the else that are inside its { }
         to execute."] ) ),

new LearningStep(
  65,
  "Now the code in the if\n  will execute, and the\n  else
       will be **skipped**.\n \nWhy will the **else** code
       in { } be **skipped**?\n\n
  The language designer decided this is what it should do for
       if - else statements. It's another of the many
       simple, independent rules you can remember and
       internalize to learn programming : )",
  undefined ),

new LearningStep(65, undefined, undefined),

new LearningStep(66, undefined, undefined),

new LearningStep(67, undefined, undefined),

new LearningStep(68, undefined, undefined),

new LearningStep(69, undefined, undefined),

new LearningStep(70, undefined, undefined),

new LearningStep(71, undefined, undefined),

new LearningStep(72, undefined, undefined),

new LearningStep(73, undefined, undefined),

new LearningStep(74, undefined, undefined),

new LearningStep(75, undefined, undefined),

new LearningStep(76, undefined, undefined),

new LearningStep(77, undefined, undefined),

new LearningStep(78, undefined, undefined),

new LearningStep(80, undefined, undefined),

new LearningStep(81, undefined, undefined),

new LearningStep(
  81,
  undefined,

  new Question(
    "What value is on the stack?",
    [1,0,-1],
    ["x was 1 in the beginning, but x > 0 but true on the
         stack. What did the if statement do then?","","The
         else statement is only executed when the condition
         is false."] ) ),

new LearningStep(
  81,
  "Great! Let's go!",
  undefined )]
                    ) /* end LearningSteps */
                    ) /* end Action */,
new Action (
                    new Program(`

x = 0;
if ( x == 0 ){
    x = 10;
} else if ( x == 2){
    x = 100;
} else {
    x = 1000;
}
x;

x = 2;
if ( x == 0 ){
    x = 1;
} else if ( x == 2){
    x = 3;
} else {
    x = 0;
}
x;

x = 3;
if ( x == 0 ){
    x = 1;
} else if ( x == 2){
    x = 3;
} else {
    x = 0;
```

```
}
x;

x = -1;
if ( x == 0 ){
    x = 1;
} else if ( x == 2){
    x = 3;
} else {
    x = 0;
}
x;

x = 0;
var y = 1;
var z = 0;
if ( x > y ){
    z = 1;
} else if ( y == x){
    z = 10;
} else {
    z = 100;
}

x;
y;
z;



x = 0;
y = 1;
z = 0;
if ( x > y ){
    z = 1;
} else if ( y == x){
    z = 10;
}



            `
            ) /* end Program */,
                    new LearningSteps(
        /* Learning Steps */
        [
  new LearningStep(
    0,
    "## If and Else If\n\n
    You can connect many ifs together into one statement by
         putting **else if** in between. The computer will
         check each condition and execute the code in the
         first one that **is** true. By is true, I mean it **
         evaluates** to true. Then it skips all the other else
          parts.\n\n
    If no conditions are true, it keeps going to the rest of
         the program.\n",
    undefined ),

  new LearningStep(0, undefined, undefined),

  new LearningStep(1, undefined, undefined),

  new LearningStep(2, undefined, undefined),

  new LearningStep(3, undefined, undefined),

  new LearningStep(4, undefined, undefined),

  new LearningStep(5, undefined, undefined),

  new LearningStep(6, undefined, undefined),

  new LearningStep(7, undefined, undefined),

  new LearningStep(8, undefined, undefined),

  new LearningStep(9, undefined, undefined),

  new LearningStep(10, undefined, undefined),

  new LearningStep(11, undefined, undefined),

  new LearningStep(13, undefined, undefined),

  new LearningStep(
    14,
    undefined,

    new Question(
      "What value is on the stack?",
      [0,3,1000],
      ["x started out as 0 but the condition for the first part
           of the if statement was true.","the else if only
          gets executed if it's condition (x ==2) is true and
           all the if's (or else ifs) before it had false
          conditions.","The else's instructions didn't
          execute because the first if condition x==0 was
          true. The else is only execute when all the
          preceding if statement conditions are false."] ) ),

  new LearningStep(15, undefined, undefined),

  new LearningStep(16, undefined, undefined),

  new LearningStep(17, undefined, undefined),

  new LearningStep(
    17,
    "What do you think this next if statement will do? Read
         through it and think, then step through it.",
    undefined ),

  new LearningStep(18, undefined, undefined),

  new LearningStep(19, undefined, undefined),

  new LearningStep(20, undefined, undefined),

  new LearningStep(21, undefined, undefined),

  new LearningStep(22, undefined, undefined),

  new LearningStep(23, undefined, undefined),

  new LearningStep(24, undefined, undefined),

  new LearningStep(25, undefined, undefined),

  new LearningStep(26, undefined, undefined),

  new LearningStep(27, undefined, undefined),

  new LearningStep(28, undefined, undefined),

  new LearningStep(29, undefined, undefined),

  new LearningStep(30, undefined, undefined),

  new LearningStep(31, undefined, undefined),

  new LearningStep(33, undefined, undefined),

  new LearningStep(34, undefined, undefined),

  new LearningStep(
    34,
    undefined,

    new Question(
      "What value is on the stack?",
      [100,1,2],
      ["","the condition for the first part of the if (x==0)
           was not true. So that code was not executed. The
           computer then goes to the next statement and finds
           the else if.","x started out as 2 but the else if (
           x == 2) had it's body executed (the code between
           its { })."] ) ),

  new LearningStep(35, undefined, undefined),

  new LearningStep(36, undefined, undefined),

  new LearningStep(37, undefined, undefined),

  new LearningStep(
    37,
    "What do you think this next if statement will do? Read
         through it and think, then step through it.",
```

```
  undefined ),

new LearningStep(38, undefined, undefined),

new LearningStep(39, undefined, undefined),

new LearningStep(40, undefined, undefined),

new LearningStep(41, undefined, undefined),

new LearningStep(42, undefined, undefined),

new LearningStep(43, undefined, undefined),

new LearningStep(44, undefined, undefined),

new LearningStep(45, undefined, undefined),

new LearningStep(46, undefined, undefined),

new LearningStep(47, undefined, undefined),

new LearningStep(48, undefined, undefined),

new LearningStep(49, undefined, undefined),

new LearningStep(50, undefined, undefined),

new LearningStep(51, undefined, undefined),

new LearningStep(52, undefined, undefined),

new LearningStep(
  52,
  undefined,

  new Question(
    "What value is on the stack?",
    [3,10,5],
    ["x started out as 3. Because both the if and else if had
          false conditions, the computer executed the final
         else's code inside its { }","",""] ) ),

new LearningStep(53, undefined, undefined),

new LearningStep(54, undefined, undefined),

new LearningStep(55, undefined, undefined),

new LearningStep(56, undefined, undefined),

new LearningStep(57, undefined, undefined),

new LearningStep(58, undefined, undefined),

new LearningStep(59, undefined, undefined),

new LearningStep(60, undefined, undefined),

new LearningStep(61, undefined, undefined),

new LearningStep(62, undefined, undefined),

new LearningStep(63, undefined, undefined),

new LearningStep(64, undefined, undefined),

new LearningStep(65, undefined, undefined),

new LearningStep(66, undefined, undefined),

new LearningStep(67, undefined, undefined),

new LearningStep(68, undefined, undefined),

new LearningStep(69, undefined, undefined),

new LearningStep(70, undefined, undefined),

new LearningStep(71, undefined, undefined),

new LearningStep(
  71,
  undefined,

  new Question(
    "What value is on the stack?",
    [-1,100,10],
    ["x started out as -1. Because both the if and else if
         had false conditions, the computer executed the
         final else's code inside its { }","",""] ) ),

new LearningStep(72, undefined, undefined),

new LearningStep(73, undefined, undefined),

new LearningStep(74, undefined, undefined),

new LearningStep(75, undefined, undefined),

new LearningStep(76, undefined, undefined),

new LearningStep(77, undefined, undefined),

new LearningStep(78, undefined, undefined),

new LearningStep(79, undefined, undefined),

new LearningStep(80, undefined, undefined),

new LearningStep(81, undefined, undefined),

new LearningStep(82, undefined, undefined),

new LearningStep(83, undefined, undefined),

new LearningStep(84, undefined, undefined),

new LearningStep(85, undefined, undefined),

new LearningStep(86, undefined, undefined),

new LearningStep(87, undefined, undefined),

new LearningStep(88, undefined, undefined),

new LearningStep(89, undefined, undefined),

new LearningStep(90, undefined, undefined),

new LearningStep(91, undefined, undefined),

new LearningStep(92, undefined, undefined),

new LearningStep(93, undefined, undefined),

new LearningStep(94, undefined, undefined),

new LearningStep(95, undefined, undefined),

new LearningStep(96, undefined, undefined),

new LearningStep(97, undefined, undefined),

new LearningStep(98, undefined, undefined),

new LearningStep(99, undefined, undefined),

new LearningStep(
  99,
  undefined,

  new Question(
    "What value is on the stack?",
    [10,100,5],
    ["x doesn't change. There is not any code that looks like
          x = something.","x doesn't change. There is not
         any code that looks like x = something.","x doesn't
          change. There is not any code that looks like x =
         something."] ) ),

new LearningStep(100, undefined, undefined),

new LearningStep(101, undefined, undefined),

new LearningStep(
  101,
  undefined,

  new Question(
```

```
    "What value is on the stack?",
    [0,100,5],
    ["y doesn't change. There is not any code that looks like
         y = something.","y doesn't change. There is not
        any code that looks like y = something.","y doesn't
         change. There is not any code that looks like y =
        something."] ) ),

new LearningStep(102, undefined, undefined),

new LearningStep(103, undefined, undefined),

new LearningStep(
  103,
  undefined,

  new Question(
    "What value is on the stack?",
    [0,1,5],
    ["This is what z started out with. It changed - why? Keep
         trying!","This is what z started out with. x > y
        was 0 > 0 , is that true?",""] ) ),

new LearningStep(
  104,
  "Sometimes all the instructions inside the { } 's get
       jumped over. The code inside the { }'s is called the
       **body** of the if statement.",
  undefined ),

new LearningStep(104, undefined, undefined),

new LearningStep(105, undefined, undefined),

new LearningStep(106, undefined, undefined),

new LearningStep(107, undefined, undefined),

new LearningStep(108, undefined, undefined),

new LearningStep(109, undefined, undefined),

new LearningStep(110, undefined, undefined),

new LearningStep(111, undefined, undefined),

new LearningStep(112, undefined, undefined),

new LearningStep(113, undefined, undefined),

new LearningStep(114, undefined, undefined),

new LearningStep(115, undefined, undefined),

new LearningStep(116, undefined, undefined),

new LearningStep(117, undefined, undefined),

new LearningStep(118, undefined, undefined),

new LearningStep(119, undefined, undefined),

new LearningStep(120, undefined, undefined),

new LearningStep(121, undefined, undefined),

new LearningStep(122, undefined, undefined),

new LearningStep(
  123,
  "That was long and\n                                    you
       stuck with it!\n                              If statements
        are hard\n                             for almost
       everyone.\n                  But they really just\n
                  mirror something you\n        do every day
       . Decide\n        to do something or\n
       something else.\n        Onwards!",
  undefined ),

new LearningStep(
  123,
  "That was long and\n                                    you
       stuck with it!\n                              If statements
        are hard\n                             for almost
       everyone.\n                  But they really just\n
                  mirror something you\n        do every day
       . Decide\n        to do something or\n
       something else.\n        Onwards!",
    undefined )]
                  ) /* end LearningSteps */
                  ) /* end Action */,
new Action (
                  new Program(`

var rooms = [ [12, 10, 8],
              [20, 12, 8]];

var room1 = rooms[0];
var room2 = rooms[1];

var room_volumes = [room1[0] * room1[1] * room1[2],
                    room2[0] * room2[1] * room2[2]];



var ac_1 = 2000;
var ac_2 = 4000;
var ac_3 = 7500;
var best_ac = -1;

var total_volume = room_volumes[0] + room_volumes[1];

if (total_volume <= ac_1){
    best_ac = 1;
}
else if (total_volume <= ac_2){
    best_ac = 2;
}
else if (total_volume <= ac_3){
    best_ac = 3;
}

best_ac;



            `
            ) /* end Program */,
                  new LearningSteps(
        /* Learning Steps */
        [
  new LearningStep(
    0,
    "## Review\n\n
    Let's integrate what\nwe've learned by continuing\nthe Air
         Conditioner Recommender\nprogram!\n\n
    Here's where we left off:\nWe made an array with \neach
         room's length, width, height.",
    undefined ),

  new LearningStep(0, undefined, undefined),

  new LearningStep(1, undefined, undefined),

  new LearningStep(2, undefined, undefined),

  new LearningStep(3, undefined, undefined),

  new LearningStep(4, undefined, undefined),

  new LearningStep(5, undefined, undefined),

  new LearningStep(6, undefined, undefined),

  new LearningStep(7, undefined, undefined),

  new LearningStep(8, undefined, undefined),

  new LearningStep(9, undefined, undefined),

  new LearningStep(10, undefined, undefined),

  new LearningStep(11, undefined, undefined),

  new LearningStep(12, undefined, undefined),

  new LearningStep(13, undefined, undefined),

  new LearningStep(14, undefined, undefined),

  new LearningStep(15, undefined, undefined),
```

```
new LearningStep(16, undefined, undefined),
new LearningStep(17, undefined, undefined),
new LearningStep(18, undefined, undefined),
new LearningStep(19, undefined, undefined),
new LearningStep(20, undefined, undefined),
new LearningStep(21, undefined, undefined),
new LearningStep(22, undefined, undefined),
new LearningStep(23, undefined, undefined),
new LearningStep(24, undefined, undefined),
new LearningStep(25, undefined, undefined),
new LearningStep(26, undefined, undefined),
new LearningStep(27, undefined, undefined),
new LearningStep(28, undefined, undefined),
new LearningStep(29, undefined, undefined),
new LearningStep(30, undefined, undefined),
new LearningStep(31, undefined, undefined),
new LearningStep(32, undefined, undefined),
new LearningStep(33, undefined, undefined),
new LearningStep(34, undefined, undefined),
new LearningStep(35, undefined, undefined),
new LearningStep(36, undefined, undefined),
new LearningStep(37, undefined, undefined),
new LearningStep(38, undefined, undefined),
new LearningStep(39, undefined, undefined),
new LearningStep(40, undefined, undefined),
new LearningStep(41, undefined, undefined),
new LearningStep(42, undefined, undefined),
new LearningStep(43, undefined, undefined),
new LearningStep(44, undefined, undefined),
new LearningStep(45, undefined, undefined),
new LearningStep(46, undefined, undefined),
new LearningStep(47, undefined, undefined),
new LearningStep(48, undefined, undefined),
new LearningStep(49, undefined, undefined),
new LearningStep(50, undefined, undefined),
new LearningStep(51, undefined, undefined),
new LearningStep(52, undefined, undefined),
new LearningStep(53, undefined, undefined),
new LearningStep(54, undefined, undefined),
new LearningStep(55, undefined, undefined),
new LearningStep(56, undefined, undefined),
new LearningStep(57, undefined, undefined),
new LearningStep(
  58,
  "Now we want to recommend\nthe cheapest air conditioner\
       nthat can handle that volume.\n\n
  Let's assume there are three\ndifferent air conditioners to
       choose from. We'll store the maximum volume each can
       handle.",
  undefined ),
new LearningStep(58, undefined, undefined),
new LearningStep(59, undefined, undefined),
new LearningStep(60, undefined, undefined),
new LearningStep(61, undefined, undefined),
new LearningStep(62, undefined, undefined),
new LearningStep(63, undefined, undefined),
new LearningStep(64, undefined, undefined),
new LearningStep(65, undefined, undefined),
new LearningStep(66, undefined, undefined),
new LearningStep(67, undefined, undefined),
new LearningStep(68, undefined, undefined),
new LearningStep(69, undefined, undefined),
new LearningStep(70, undefined, undefined),
new LearningStep(71, undefined, undefined),
new LearningStep(
  72,
  "We'll also make a variable\nto store the number of\nthe
       air conditioner that\nis best. We'll put -1\nin there
       to start.\nIf people see the program\nrecommend -1,
       they can\nread this comment to know\nthat is what -1
       means.",
  undefined ),
new LearningStep(73, undefined, undefined),
new LearningStep(
  73,
  "We'll also make a variable\nto store the number of\nthe
       air conditioner that\nis best. We'll put -1\nin there
       to start.\nIf people see the program\nrecommend -1,
       it means none of them will work.",
  undefined ),
new LearningStep(74, undefined, undefined),
new LearningStep(75, undefined, undefined),
new LearningStep(76, undefined, undefined),
new LearningStep(77, undefined, undefined),
new LearningStep(78, undefined, undefined),
new LearningStep(79, undefined, undefined),
new LearningStep(80, undefined, undefined),
new LearningStep(81, undefined, undefined),
new LearningStep(82, undefined, undefined),
new LearningStep(83, undefined, undefined),
new LearningStep(84, undefined, undefined),
new LearningStep(85, undefined, undefined),
new LearningStep(86, undefined, undefined),
new LearningStep(87, undefined, undefined),
new LearningStep(88, undefined, undefined),
```

```
  new LearningStep(89, undefined, undefined),

  new LearningStep(
    90,
    "Let's use if's to recommend an air conditioner. Before we
         step through it, you should **mentally step through
         it**. Read the code yourself and **look at the
         namespace** to look up the values of the variables.
         What will the best air conditioner be? ",
    undefined ),

  new LearningStep(
    90,
    "Did you really do that? Get up and take a break for a
         couple minutes if you're feeling fatigued - walk
         around, drink some water, a little snack, they all
         help you learn.",
    undefined ),

  new LearningStep(91, undefined, undefined),

  new LearningStep(92, undefined, undefined),

  new LearningStep(93, undefined, undefined),

  new LearningStep(94, undefined, undefined),

  new LearningStep(95, undefined, undefined),

  new LearningStep(96, undefined, undefined),

  new LearningStep(97, undefined, undefined),

  new LearningStep(98, undefined, undefined),

  new LearningStep(99, undefined, undefined),

  new LearningStep(100, undefined, undefined),

  new LearningStep(101, undefined, undefined),

  new LearningStep(102, undefined, undefined),

  new LearningStep(103, undefined, undefined),

  new LearningStep(104, undefined, undefined),

  new LearningStep(105, undefined, undefined),

  new LearningStep(106, undefined, undefined),

  new LearningStep(
    106,
    undefined,

    new Question(
      "What was the number of the best one? ",
      [0,10,100],
      ["","","",""] ) ),

  new LearningStep(107, undefined, undefined)]
                    ) /* end LearningSteps */
                    ) /* end Action */]),
new KnowledgeUnit("Operators Review",
            "Operators use values on the stack to do a
                 calculation.
            They can use number values or boolean values. There
                  are only two boolean values - true and false
                .
            + - * / > < >= <= == != work on numbers. ! != == &&
                 || work on booleans.
            The parser uses some rules to decide what
            order to do the operators in. A simple rule to
        figure out if you are NOT doing
        the operators in the right order:

            If you find yourself comparing
        a number and a true/false (boolean)
        value (for example, true > 12 , or
            12 && false), you're doing the
        operators in the wrong order.

            If you get confused, try
        reading the expression out loud.",
                    [
new Action (
                        new Program(`


var x_1 = 1 > 2 && 3 > 4;
var x_2 = 2 > 1 && 3 >= 3;

var a = 10;
var x_3 = a >= 9 || a < 9;



a == 5 && 10 > 8;

var x_4 = a + 1 > 10;
var x_4p = 10 < a + 1;
var x_5 = 10 <= a - 1;
var x_6 = 1 > 0;

var x_7 = 1 <= 0;
var x_8 = 1 < 0;
var x_9 = 0 <= 1;
var x_10 = 0 < 1;
var x_11 = 0 <= 0;
var x_12 = 0 < 0;




            `
            ) /* end Program */,
                        new LearningSteps(
        /* Learning Steps */
        [
  new LearningStep(
    0,
    "Here are all the operators we've taught you about in
         Javascript.\nWe'll review all this information again
         as we go through them\n\n
    Binary Operator        \n1. (left) + (right)       Addition
         \n2. (left) - (right)       Subtraction\n3. (left) *
         (right)       Multiplication\n4. (left) / (right)
              Division\n5. (left) = (right)       assignment
         \n\n
    All of the operators above put a number value onto the
         stack.\n\n
    All of the operators below put a boolean value onto the
         stack.\n\n
    Binary Operator        \n1. (left) > (right)       greater
         than\n2. (left) < (right)       less than\n3. (left)
         == (right)      equal to\n4. (left) >= (right)
         greater than or equal to\n5. (left) <= (right)
         less than or equal to\n6. (left) != (right)      not
         equal to\n\n
    Here are two more boolean operators that are very\
         nimportant but the language designers chose to use\
         nsome strange symbols to represent them.\n\n
    **and**\n```\n(left) && (right)\n```\nThis operator leaves
         true on the stack\nif both left and right true.\
         nOtherwise it leaves false.\n\n
    **or** \n```\n(left) || (right)\n```   \nThis operator
         leaves true on the stack if at least one of left or
         right is true.\nOtherwise it leaves false.",
    undefined ),

  new LearningStep(0, undefined, undefined),

  new LearningStep(1, undefined, undefined),

  new LearningStep(2, undefined, undefined),

  new LearningStep(3, undefined, undefined),

  new LearningStep(4, undefined, undefined),

  new LearningStep(5, undefined, undefined),

  new LearningStep(6, undefined, undefined),

  new LearningStep(7, undefined, undefined),

  new LearningStep(8, undefined, undefined),
```

```
new LearningStep(9, undefined, undefined),

new LearningStep(
  9,
  undefined,

  new Question(
    "What is x_1s value after this instruction executes?",
    [true,[false],[true]],
    ["Those are greater than symbols. && means and","The
         answer is not an array, just a boolean value.","The
          answer is not an array, just a boolean value."] )
         ),

new LearningStep(10, undefined, undefined),

new LearningStep(11, undefined, undefined),

new LearningStep(12, undefined, undefined),

new LearningStep(13, undefined, undefined),

new LearningStep(14, undefined, undefined),

new LearningStep(15, undefined, undefined),

new LearningStep(16, undefined, undefined),

new LearningStep(17, undefined, undefined),

new LearningStep(18, undefined, undefined),

new LearningStep(19, undefined, undefined),

new LearningStep(20, undefined, undefined),

new LearningStep(21, undefined, undefined),

new LearningStep(
  21,
  undefined,

  new Question(
    "What is x_2s value after this instruction executes?",
    [false,[false],[true]],
    ["","The answer is not an array, just a boolean value.","
         The answer is not an array, just a boolean value."]
          ) ),

new LearningStep(22, undefined, undefined),

new LearningStep(23, undefined, undefined),

new LearningStep(24, undefined, undefined),

new LearningStep(25, undefined, undefined),

new LearningStep(26, undefined, undefined),

new LearningStep(27, undefined, undefined),

new LearningStep(28, undefined, undefined),

new LearningStep(29, undefined, undefined),

new LearningStep(30, undefined, undefined),

new LearningStep(31, undefined, undefined),

new LearningStep(32, undefined, undefined),

new LearningStep(33, undefined, undefined),

new LearningStep(34, undefined, undefined),

new LearningStep(35, undefined, undefined),

new LearningStep(36, undefined, undefined),

new LearningStep(37, undefined, undefined),

new LearningStep(
  37,
  undefined,

  new Question(
    "What is x_3s value after this instruction executes?",
    [false,[true],[false]],
    ["|| means or.","The answer is not an array, just a
         boolean value.","The answer is not an array, just a
          boolean value."] ) ),

new LearningStep(
  38,
  "When there are multiple operators\nhow does the computer
       decide\nwhat order to do them in?\n\n
  The parser uses some rules to decide what\norder to do the
       operators in. We won't\nteach you all the details
       because\nthe rules were designed\nto work in the way
       most people expect.\n\n
  We can teach you a simple rule to\nfigure out if you are
       NOT doing\nthe operators in the right order.\nHere it
        is:\n\n
  If you find yourself comparing\na number and a true/false (
       boolean)\nvalue (for example, true > 12 , or\n12 &&
       false), you're doing the\noperators in the wrong
       order.\n\n
  If you get confused, try\nreading the expression out loud.\
       nFor example,\n\n
  a == 5 && 10 > 8 \ncan be read as\nx equals 5 and 10 is
       greater than 8",
  undefined ),

new LearningStep(38, undefined, undefined),

new LearningStep(39, undefined, undefined),

new LearningStep(40, undefined, undefined),

new LearningStep(41, undefined, undefined),

new LearningStep(42, undefined, undefined),

new LearningStep(43, undefined, undefined),

new LearningStep(44, undefined, undefined),

new LearningStep(45, undefined, undefined),

new LearningStep(
  45,
  undefined,

  new Question(
    "What value is on the stack before this instruction
         executes?",
    [true,[true],[false]],
    ["What was a's value? && means and, so both sides must be
         true for it to put true on the stack.","The answer
         is not an array, just a boolean value.","The
        answer is not an array, just a boolean value."] ) )
        ,

new LearningStep(46, undefined, undefined),

new LearningStep(47, undefined, undefined),

new LearningStep(48, undefined, undefined),

new LearningStep(49, undefined, undefined),

new LearningStep(50, undefined, undefined),

new LearningStep(51, undefined, undefined),

new LearningStep(52, undefined, undefined),

new LearningStep(53, undefined, undefined),

new LearningStep(54, undefined, undefined),

new LearningStep(
  54,
  undefined,

  new Question(
    "What is x_4s value after this instruction executes?",
    [false,[true],[false]],
    ["","The answer is not an array, just a boolean value.","
         The answer is not an array, just a boolean value."]
```

```
        ) ),

new LearningStep(55, undefined, undefined),

new LearningStep(56, undefined, undefined),

new LearningStep(57, undefined, undefined),

new LearningStep(58, undefined, undefined),

new LearningStep(59, undefined, undefined),

new LearningStep(60, undefined, undefined),

new LearningStep(61, undefined, undefined),

new LearningStep(62, undefined, undefined),

new LearningStep(63, undefined, undefined),

new LearningStep(
  63,
  undefined,

  new Question(
    "What is x_4ps value after this instruction executes?"
        ,
    [false,[true],[false]],
    ["","The answer is not an array, just a boolean value.","
        The answer is not an array, just a boolean value."]
    ) ),

new LearningStep(64, undefined, undefined),

new LearningStep(65, undefined, undefined),

new LearningStep(66, undefined, undefined),

new LearningStep(67, undefined, undefined),

new LearningStep(68, undefined, undefined),

new LearningStep(69, undefined, undefined),

new LearningStep(70, undefined, undefined),

new LearningStep(71, undefined, undefined),

new LearningStep(72, undefined, undefined),

new LearningStep(
  72,
  undefined,

  new Question(
    "What is x_5s value after this instruction executes?",
    [true,[true],[false]],
    ["","The answer is not an array, just a boolean value.","
        The answer is not an array, just a boolean value."]
    ) ),

new LearningStep(73, undefined, undefined),

new LearningStep(74, undefined, undefined),

new LearningStep(75, undefined, undefined),

new LearningStep(76, undefined, undefined),

new LearningStep(77, undefined, undefined),

new LearningStep(78, undefined, undefined),

new LearningStep(79, undefined, undefined),

new LearningStep(
  79,
  undefined,

  new Question(
    "What is x_6s value after this instruction executes?",
    [false,[false],[true]],
    ["","The answer is not an array, just a boolean value.","
        The answer is not an array, just a boolean value."]
    ) ),

new LearningStep(80, undefined, undefined),

new LearningStep(81, undefined, undefined),

new LearningStep(82, undefined, undefined),

new LearningStep(83, undefined, undefined),

new LearningStep(84, undefined, undefined),

new LearningStep(85, undefined, undefined),

new LearningStep(86, undefined, undefined),

new LearningStep(
  86,
  undefined,

  new Question(
    "What is x_7s value after this instruction executes?",
    [true,[true],[false]],
    ["","The answer is not an array, just a boolean value.","
        The answer is not an array, just a boolean value."]
    ) ),

new LearningStep(87, undefined, undefined),

new LearningStep(88, undefined, undefined),

new LearningStep(89, undefined, undefined),

new LearningStep(90, undefined, undefined),

new LearningStep(91, undefined, undefined),

new LearningStep(92, undefined, undefined),

new LearningStep(93, undefined, undefined),

new LearningStep(
  93,
  undefined,

  new Question(
    "What is x_8s value after this instruction executes?",
    [true,[true],[false]],
    ["","The answer is not an array, just a boolean value.","
        The answer is not an array, just a boolean value."]
    ) ),

new LearningStep(94, undefined, undefined),

new LearningStep(95, undefined, undefined),

new LearningStep(96, undefined, undefined),

new LearningStep(97, undefined, undefined),

new LearningStep(98, undefined, undefined),

new LearningStep(99, undefined, undefined),

new LearningStep(100, undefined, undefined),

new LearningStep(
  100,
  undefined,

  new Question(
    "What is x_9s value after this instruction executes?",
    [false,[true],[false]],
    ["","The answer is not an array, just a boolean value.","
        The answer is not an array, just a boolean value."]
    ) ),

new LearningStep(101, undefined, undefined),

new LearningStep(102, undefined, undefined),

new LearningStep(103, undefined, undefined),

new LearningStep(104, undefined, undefined),

new LearningStep(105, undefined, undefined),
```

```
  new LearningStep(106, undefined, undefined),

  new LearningStep(107, undefined, undefined),

  new LearningStep(
    107,
    undefined,

    new Question(
      "What is x_10's value after this instruction executes?"
          ,
      [false,[true],[false]],
      ["","The answer is not an array, just a boolean value.","
          The answer is not an array, just a boolean value."]
       )),

  new LearningStep(108, undefined, undefined),

  new LearningStep(109, undefined, undefined),

  new LearningStep(110, undefined, undefined),

  new LearningStep(111, undefined, undefined),

  new LearningStep(112, undefined, undefined),

  new LearningStep(113, undefined, undefined),

  new LearningStep(114, undefined, undefined),

  new LearningStep(
    114,
    undefined,

    new Question(
      "What is x_11's value after this instruction executes?"
          ,
      [false,[false],[true]],
      ["","The answer is not an array, just a boolean value.","
          The answer is not an array, just a boolean value."]
       )),

  new LearningStep(115, undefined, undefined),

  new LearningStep(116, undefined, undefined),

  new LearningStep(117, undefined, undefined),

  new LearningStep(118, undefined, undefined),

  new LearningStep(119, undefined, undefined),

  new LearningStep(120, undefined, undefined),

  new LearningStep(121, undefined, undefined),

  new LearningStep(
    121,
    undefined,

    new Question(
      "What is x_12's value after this instruction executes?"
          ,
      [true,[false],[true]],
      ["","The answer is not an array, just a boolean value.","
          The answer is not an array, just a boolean value."]
       ))]
              ) /* end LearningSteps */
              ) /* end Action */]),
new KnowledgeUnit("Loops Enable Repeating",
          "Loops allow the computer to do something over and
               over. Language designers made 3 kinds of
               loops:while, for, and do-while. They mostly
               work the same - they repeat some instructions
                if a condition is true. Loops allow programs
                to be flexible and work with arrays of
               different sizes.",
          [
new Action (
              new Program(`
var x = 0;
if ( x <= 1 ) {
    x = x + 1;
}
if ( x <= 1 ) {
    x = x + 1;
}
if ( x <= 1 ) {
    x = x + 1;
}
x;


x = 0;
while ( x <= 1 ){
    x = x + 1;
}
x;

x = 0;
while ( x <= 2 ){
    x = x + 1;
}
x;

x = 0;
while ( x <= 0 ){
    x = x + 1;
}
x;

x = 10;
while ( x <= 0 ){
    x = x + 1;
}
x;
          `
          ) /* end Program */,
                new LearningSteps(
        /* Learning Steps */
        [
  new LearningStep(
    0,
    "## While\n\n
    You use a **while** statement to execute some code over and
         over. The syntax is:\n```\nwhile ( condition ) {\n
        code to repeat\n}\n``` \n\n
    Like an if statement, it uses the **condition** expression
         to decide to keep going or not. We think the word
         repeat instead of while would have been more clear
         for people, but we didn't design this language.\n",
    undefined ),

  new LearningStep(
    0,
    "## Why While\n\n
    While statements can make code **shorter** and also more **
         flexible**.\n\n
    Here's an example of code that takes 3 if statements. We'll
          follow it with a while statement that **effectively
         ** does the same thing.\n",
    undefined ),

  new LearningStep(0, undefined, undefined),

  new LearningStep(1, undefined, undefined),

  new LearningStep(2, undefined, undefined),

  new LearningStep(3, undefined, undefined),

  new LearningStep(4, undefined, undefined),

  new LearningStep(5, undefined, undefined),

  new LearningStep(6, undefined, undefined),

  new LearningStep(7, undefined, undefined),

  new LearningStep(8, undefined, undefined),

  new LearningStep(9, undefined, undefined),
```

```
new LearningStep(10, undefined, undefined),

new LearningStep(11, undefined, undefined),

new LearningStep(12, undefined, undefined),

new LearningStep(13, undefined, undefined),

new LearningStep(14, undefined, undefined),

new LearningStep(15, undefined, undefined),

new LearningStep(
  15,
  undefined,

  new Question(
    "What value does x have?",
    [5,10,100],
    ["","",""] ) ),

new LearningStep(16, undefined, undefined),

new LearningStep(17, undefined, undefined),

new LearningStep(18, undefined, undefined),

new LearningStep(19, undefined, undefined),

new LearningStep(20, undefined, undefined),

new LearningStep(21, undefined, undefined),

new LearningStep(22, undefined, undefined),

new LearningStep(23, undefined, undefined),

new LearningStep(24, undefined, undefined),

new LearningStep(25, undefined, undefined),

new LearningStep(
  25,
  undefined,

  new Question(
    " What value does x have?",
    [0,100,5],
    ["","",""] ) ),

new LearningStep(26, undefined, undefined),

new LearningStep(27, undefined, undefined),

new LearningStep(28, undefined, undefined),

new LearningStep(29, undefined, undefined),

new LearningStep(30, undefined, undefined),

new LearningStep(31, undefined, undefined),

new LearningStep(32, undefined, undefined),

new LearningStep(33, undefined, undefined),

new LearningStep(34, undefined, undefined),

new LearningStep(
  34,
  "Step through the **while** statement carefully and read
       the descriptions.",
  undefined ),

new LearningStep(35, undefined, undefined),

new LearningStep(36, undefined, undefined),

new LearningStep(37, undefined, undefined),

new LearningStep(38, undefined, undefined),

new LearningStep(39, undefined, undefined),

new LearningStep(
  39,
  "Because the condition is true, the **body** of the while
       loop - the code inside its { } - will execute.",
  undefined ),

new LearningStep(40, undefined, undefined),

new LearningStep(41, undefined, undefined),

new LearningStep(42, undefined, undefined),

new LearningStep(43, undefined, undefined),

new LearningStep(44, undefined, undefined),

new LearningStep(
  44,
  undefined,

  new Question(
    "What value does x have?",
    [10,5,0],
    ["","","x started out as zero but the body of the while
         loop just executed"] ) ),

new LearningStep(
  44,
  "Now, the computer follows the rule for a while statement
       and **goes to check the condition again**.",
  undefined ),

new LearningStep(45, undefined, undefined),

new LearningStep(46, undefined, undefined),

new LearningStep(47, undefined, undefined),

new LearningStep(48, undefined, undefined),

new LearningStep(49, undefined, undefined),

new LearningStep(
  49,
  "Because the condition is true, the while loop's body will
       execute again.",
  undefined ),

new LearningStep(50, undefined, undefined),

new LearningStep(51, undefined, undefined),

new LearningStep(52, undefined, undefined),

new LearningStep(53, undefined, undefined),

new LearningStep(54, undefined, undefined),

new LearningStep(
  54,
  "Now, the computer follows the rule for a while statement
       and **goes to check the condition again** when it
       reaches the end of the body.",
  undefined ),

new LearningStep(55, undefined, undefined),

new LearningStep(56, undefined, undefined),

new LearningStep(57, undefined, undefined),

new LearningStep(58, undefined, undefined),

new LearningStep(59, undefined, undefined),

new LearningStep(
  59,
  "Now, because the condition was false (since x <= 1 left
       false on the stack), the while **skips over its body
       ** (just like an if statement does when the condition
        is false).",
  undefined ),

new LearningStep(60, undefined, undefined),

new LearningStep(61, undefined, undefined),
```

```
new LearningStep(
  61,
  undefined,

  new Question(
    "What value is in x?",
    [1,100,5],
    ["The while loop repeated twice, go back to x = 0 and
         step through again from the beginning.","","The
         while loop only repeated twice, go back to x = 0
         and step through again from the beginning."] ) ),

new LearningStep(
  61,
  " Let's do some questions!",
  undefined ),

new LearningStep(62, undefined, undefined),

new LearningStep(63, undefined, undefined),

new LearningStep(64, undefined, undefined),

new LearningStep(65, undefined, undefined),

new LearningStep(66, undefined, undefined),

new LearningStep(67, undefined, undefined),

new LearningStep(68, undefined, undefined),

new LearningStep(69, undefined, undefined),

new LearningStep(
  69,
  "Where do you think the computer will start executing next?
       ",
  undefined ),

new LearningStep(70, undefined, undefined),

new LearningStep(71, undefined, undefined),

new LearningStep(72, undefined, undefined),

new LearningStep(73, undefined, undefined),

new LearningStep(74, undefined, undefined),

new LearningStep(75, undefined, undefined),

new LearningStep(76, undefined, undefined),

new LearningStep(77, undefined, undefined),

new LearningStep(78, undefined, undefined),

new LearningStep(79, undefined, undefined),

new LearningStep(
  79,
  undefined,

  new Question(
    "What did the condition evaluate to?",
    [false,-10,10],
    ["","The answer is a boolean, not a number.","The answer
         is a boolean, not a number."] ) ),

new LearningStep(80, undefined, undefined),

new LearningStep(81, undefined, undefined),

new LearningStep(82, undefined, undefined),

new LearningStep(83, undefined, undefined),

new LearningStep(84, undefined, undefined),

new LearningStep(85, undefined, undefined),

new LearningStep(86, undefined, undefined),

new LearningStep(87, undefined, undefined),

new LearningStep(88, undefined, undefined),

new LearningStep(89, undefined, undefined),

new LearningStep(90, undefined, undefined),

new LearningStep(91, undefined, undefined),

new LearningStep(92, undefined, undefined),

new LearningStep(93, undefined, undefined),

new LearningStep(94, undefined, undefined),

new LearningStep(95, undefined, undefined),

new LearningStep(96, undefined, undefined),

new LearningStep(97, undefined, undefined),

new LearningStep(98, undefined, undefined),

new LearningStep(
  99,
  undefined,

  new Question(
    "What did the condition evaluate to (leave on the stack)?
         ",
    [true,10,-10],
    ["","The answer is a boolean, not a number.","The answer
         is a boolean, not a number."] ) ),

new LearningStep(99, undefined, undefined),

new LearningStep(100, undefined, undefined),

new LearningStep(101, undefined, undefined),

new LearningStep(
  101,
  undefined,

  new Question(
    "What value is x?",
    [5,0,100],
    ["","",""] ) ),

new LearningStep(102, undefined, undefined),

new LearningStep(103, undefined, undefined),

new LearningStep(104, undefined, undefined),

new LearningStep(105, undefined, undefined),

new LearningStep(
  105,
  "The body of the while loop **does not have** to repeat.",
  undefined ),

new LearningStep(106, undefined, undefined),

new LearningStep(107, undefined, undefined),

new LearningStep(108, undefined, undefined),

new LearningStep(109, undefined, undefined),

new LearningStep(110, undefined, undefined),

new LearningStep(111, undefined, undefined),

new LearningStep(112, undefined, undefined),

new LearningStep(113, undefined, undefined),

new LearningStep(114, undefined, undefined),

new LearningStep(115, undefined, undefined),

new LearningStep(116, undefined, undefined),

new LearningStep(117, undefined, undefined),
```

```
  new LearningStep(118, undefined, undefined),

  new LearningStep(119, undefined, undefined),

  new LearningStep(120, undefined, undefined),

  new LearningStep(121, undefined, undefined),

  new LearningStep(
    121,
    undefined,

    new Question(
      "What is the value of x?",
      [0,10,5],
      ["","",""] ) ),

  new LearningStep(122, undefined, undefined),

  new LearningStep(123, undefined, undefined),

  new LearningStep(124, undefined, undefined),

  new LearningStep(125, undefined, undefined),

  new LearningStep(
    125,
    " Just like an if statement, a **while** statement may not
        execute its body even once.",
    undefined ),

  new LearningStep(126, undefined, undefined),

  new LearningStep(127, undefined, undefined),

  new LearningStep(128, undefined, undefined),

  new LearningStep(
    129,
    undefined,

    new Question(
      "What did the condition x<=0 evaluate to?",
      [true,-10,10],
      ["","The answer is a boolean, not a number.","The answer
          is a boolean, not a number."] ) ),

  new LearningStep(129, undefined, undefined),

  new LearningStep(
    130,
    "Since the condition was false, the **while** statement
        skipped over the code in { } (its **body**).",
    undefined ),

  new LearningStep(
    130,
    undefined,

    new Question(
      "What is the value of x?",
      [0,5,1],
      ["","",""] ) ),

  new LearningStep(130, undefined, undefined),

  new LearningStep(
    131,
    "Fantastic! Let's get some more practice.",
    undefined )]
                    ) /* end LearningSteps */
                    ) /* end Action */,
new Action (
                    new Program(`
var x = true;
var y = 0;

while ( x ) {
    y = 4;
    x = false;
}

y;
```

```
y = 1;
while ( y < 5 ){
    y = y * 2;
}

y;

            `
            ) /* end Program */,
                    new LearningSteps(
        /* Learning Steps */
        [
  new LearningStep(
    0,
    "Let's try some more!",
    undefined ),

  new LearningStep(0, undefined, undefined),

  new LearningStep(1, undefined, undefined),

  new LearningStep(2, undefined, undefined),

  new LearningStep(3, undefined, undefined),

  new LearningStep(4, undefined, undefined),

  new LearningStep(5, undefined, undefined),

  new LearningStep(6, undefined, undefined),

  new LearningStep(7, undefined, undefined),

  new LearningStep(8, undefined, undefined),

  new LearningStep(9, undefined, undefined),

  new LearningStep(10, undefined, undefined),

  new LearningStep(
    10,
    "Take a moment to read this while statement and predict
        what it will do, then step through it.",
    undefined ),

  new LearningStep(11, undefined, undefined),

  new LearningStep(12, undefined, undefined),

  new LearningStep(13, undefined, undefined),

  new LearningStep(14, undefined, undefined),

  new LearningStep(15, undefined, undefined),

  new LearningStep(16, undefined, undefined),

  new LearningStep(17, undefined, undefined),

  new LearningStep(18, undefined, undefined),

  new LearningStep(19, undefined, undefined),

  new LearningStep(20, undefined, undefined),

  new LearningStep(21, undefined, undefined),

  new LearningStep(22, undefined, undefined),

  new LearningStep(
    22,
    undefined,

    new Question(
      "What is the value of y at this point?",
      [1,2,3],
      [] ) ),

  new LearningStep(23, undefined, undefined),

  new LearningStep(24, undefined, undefined),

  new LearningStep(25, undefined, undefined),

  new LearningStep(26, undefined, undefined),
```

```
new LearningStep(
  26,
  "Now, try mentally executing the program from here until
       the next \n```\n y; \n```\nThen step through the loop
        and answer the next question.\n\n
  It's hard at first for everyone, you'll get better with
       practice : )",
  undefined ),

new LearningStep(27, undefined, undefined),

new LearningStep(28, undefined, undefined),

new LearningStep(29, undefined, undefined),

new LearningStep(30, undefined, undefined),

new LearningStep(31, undefined, undefined),

new LearningStep(
  31,
  undefined,

  new Question(
    "What did the condition evaluate to?",
    [false,-10,10],
    ["","The answer is a boolean, not a number.","The answer
         is a boolean, not a number."] ) ),

new LearningStep(32, undefined, undefined),

new LearningStep(33, undefined, undefined),

new LearningStep(34, undefined, undefined),

new LearningStep(35, undefined, undefined),

new LearningStep(36, undefined, undefined),

new LearningStep(37, undefined, undefined),

new LearningStep(
  37,
  undefined,

  new Question(
    " What value does y have now?",
    [0,10,5],
    ["","",""] ) ),

new LearningStep(38, undefined, undefined),

new LearningStep(39, undefined, undefined),

new LearningStep(40, undefined, undefined),

new LearningStep(41, undefined, undefined),

new LearningStep(
  41,
  undefined,

  new Question(
    "What did the condition evaluate to?",
    [false,10,-10],
    ["","The answer is a boolean, not a number.","The answer
         is a boolean, not a number."] ) ),

new LearningStep(42, undefined, undefined),

new LearningStep(43, undefined, undefined),

new LearningStep(44, undefined, undefined),

new LearningStep(45, undefined, undefined),

new LearningStep(46, undefined, undefined),

new LearningStep(47, undefined, undefined),

new LearningStep(
  47,
  undefined,

  new Question(
    " What value does y have?",
    [1,5,0],
    ["","",""] ) ),

new LearningStep(48, undefined, undefined),

new LearningStep(49, undefined, undefined),

new LearningStep(50, undefined, undefined),

new LearningStep(51, undefined, undefined),

new LearningStep(
  51,
  undefined,

  new Question(
    "What did the condition y<5 evaluate to?",
    [false,-10,10],
    ["","The answer is a boolean, not a number.","The answer
         is a boolean, not a number."] ) ),

new LearningStep(52, undefined, undefined),

new LearningStep(53, undefined, undefined),

new LearningStep(54, undefined, undefined),

new LearningStep(55, undefined, undefined),

new LearningStep(56, undefined, undefined),

new LearningStep(57, undefined, undefined),

new LearningStep(58, undefined, undefined),

new LearningStep(59, undefined, undefined),

new LearningStep(60, undefined, undefined),

new LearningStep(61, undefined, undefined),

new LearningStep(62, undefined, undefined),

new LearningStep(
  62,
  undefined,

  new Question(
    "What is the value of y at this point?",
    [3,5,10],
    [] ) ),

new LearningStep(63, undefined, undefined),

new LearningStep(
  63,
  "Fantastic! It takes everyone practice to get better at
       mentally executing code, and it's an important skill.
        On to the next program!",
  undefined )]
                ) /* end LearningSteps */
                ) /* end Action */,
new Action (
                new Program(`  var numbers = [1,10];
var total = 0;

total = total + numbers[0];
total = total + numbers[1];
total;

var total = 0;
var position = 0;

total = total + numbers[position];
position = position + 1;
total = total + numbers[position];
total;

var numbers = [1,10];
total = 0;
position = 0;
while ( position < numbers.length ){
  total = total + numbers[position];
```

```
  position = position + 1;
}

var numbers = [1,10];
total = 0;
i = 0;
while ( i < numbers.length ){
  total = total + numbers[i];
  i = i + 1;
}

var numbers = [1,10];
var total = 0;
i = numbers.length - 1;
while ( i >= 0 ){
  total = total + numbers[i];
  i = i - 1;
}

            `
            ) /* end Program */,
                    new LearningSteps(
        /* Learning Steps */
        [
  new LearningStep(
    0,
    "## Loops and arrays\n\n
    Loops let programmers tell computers to do things people
         would get tired or too bored to do. Like add up all
         the numbers in an array.\n\n
    We wrote a long program that has different examples of
         adding up all the numbers in an array. First we will
         show more explicit but inflexible ways. Then we will
         show a while loop that accomplishes the same goal.",
    undefined ),

  new LearningStep(0, undefined, undefined),

  new LearningStep(1, undefined, undefined),

  new LearningStep(2, undefined, undefined),

  new LearningStep(3, undefined, undefined),

  new LearningStep(4, undefined, undefined),

  new LearningStep(5, undefined, undefined),

  new LearningStep(6, undefined, undefined),

  new LearningStep(7, undefined, undefined),

  new LearningStep(8, undefined, undefined),

  new LearningStep(9, undefined, undefined),

  new LearningStep(10, undefined, undefined),

  new LearningStep(11, undefined, undefined),

  new LearningStep(12, undefined, undefined),

  new LearningStep(13, undefined, undefined),

  new LearningStep(14, undefined, undefined),

  new LearningStep(15, undefined, undefined),

  new LearningStep(16, undefined, undefined),

  new LearningStep(17, undefined, undefined),

  new LearningStep(18, undefined, undefined),

  new LearningStep(19, undefined, undefined),

  new LearningStep(
    19,
    " ",

    new Question(
      "What value does total have?",
      [100,0,10],
      ["","",""] ) ),

  new LearningStep(20, undefined, undefined),

  new LearningStep(21, undefined, undefined),

  new LearningStep(22, undefined, undefined),

  new LearningStep(23, undefined, undefined),

  new LearningStep(24, undefined, undefined),

  new LearningStep(25, undefined, undefined),

  new LearningStep(26, undefined, undefined),

  new LearningStep(
    26,
    " ",

    new Question(
      "What value does total have?",
      [5,1,10],
      ["","",""] ) ),

  new LearningStep(27, undefined, undefined),

  new LearningStep(28, undefined, undefined),

  new LearningStep(29, undefined, undefined),

  new LearningStep(
    29,
    "Now, instead of writing the index manually, we will store
         it in a variable called *position*. This isn't a
         special name, we could have called it bob or x493 if
         we wished.",
    undefined ),

  new LearningStep(30, undefined, undefined),

  new LearningStep(31, undefined, undefined),

  new LearningStep(32, undefined, undefined),

  new LearningStep(33, undefined, undefined),

  new LearningStep(34, undefined, undefined),

  new LearningStep(35, undefined, undefined),

  new LearningStep(36, undefined, undefined),

  new LearningStep(37, undefined, undefined),

  new LearningStep(38, undefined, undefined),

  new LearningStep(39, undefined, undefined),

  new LearningStep(40, undefined, undefined),

  new LearningStep(41, undefined, undefined),

  new LearningStep(
    42,
    "",

    new Question(
      "What value did position leave on the stack?",
      [1,100,5],
      ["","",""] ) ),

  new LearningStep(
    42,
    "Since 0 is on the stack, this instruction will look up
         numbers[0]. ",
    undefined ),

  new LearningStep(43, undefined, undefined),

  new LearningStep(44, undefined, undefined),

  new LearningStep(45, undefined, undefined),

  new LearningStep(46, undefined, undefined),

  new LearningStep(47, undefined, undefined),
```

```
new LearningStep(48, undefined, undefined),

new LearningStep(49, undefined, undefined),

new LearningStep(50, undefined, undefined),

new LearningStep(51, undefined, undefined),

new LearningStep(52, undefined, undefined),

new LearningStep(53, undefined, undefined),

new LearningStep(54, undefined, undefined),

new LearningStep(55, undefined, undefined),

new LearningStep(56, undefined, undefined),

new LearningStep(57, undefined, undefined),

new LearningStep(58, undefined, undefined),

new LearningStep(
  59,
  "",

  new Question(
    "What is the value of total?",
    [0,1,5],
    ["","",""] ) ),

new LearningStep(59, undefined, undefined),

new LearningStep(60, undefined, undefined),

new LearningStep(
  60,
  "Now, we will show you a while loop that accomplishes the
       same goal of adding up all the numbers in an array. "
       ,
  undefined ),

new LearningStep(61, undefined, undefined),

new LearningStep(62, undefined, undefined),

new LearningStep(63, undefined, undefined),

new LearningStep(64, undefined, undefined),

new LearningStep(65, undefined, undefined),

new LearningStep(66, undefined, undefined),

new LearningStep(67, undefined, undefined),

new LearningStep(68, undefined, undefined),

new LearningStep(69, undefined, undefined),

new LearningStep(70, undefined, undefined),

new LearningStep(71, undefined, undefined),

new LearningStep(72, undefined, undefined),

new LearningStep(73, undefined, undefined),

new LearningStep(74, undefined, undefined),

new LearningStep(75, undefined, undefined),

new LearningStep(76, undefined, undefined),

new LearningStep(77, undefined, undefined),

new LearningStep(
  78,
  "",

  new Question(
    "What did numbers.length leave on the stack?\n",
    [0,5,100],
    ["","",""] ) ),

new LearningStep(78, undefined, undefined),

new LearningStep(79, undefined, undefined),

new LearningStep(80, undefined, undefined),

new LearningStep(81, undefined, undefined),

new LearningStep(82, undefined, undefined),

new LearningStep(83, undefined, undefined),

new LearningStep(84, undefined, undefined),

new LearningStep(85, undefined, undefined),

new LearningStep(86, undefined, undefined),

new LearningStep(87, undefined, undefined),

new LearningStep(88, undefined, undefined),

new LearningStep(89, undefined, undefined),

new LearningStep(90, undefined, undefined),

new LearningStep(91, undefined, undefined),

new LearningStep(92, undefined, undefined),

new LearningStep(
  92,
  "",

  new Question(
    "What is the value of total now?\n",
    [10,5,100],
    ["","",""] ) ),

new LearningStep(
  92,
  "",

  new Question(
    "What is the value of position?",
    [10,0,100],
    ["","",""] ) ),

new LearningStep(93, undefined, undefined),

new LearningStep(94, undefined, undefined),

new LearningStep(95, undefined, undefined),

new LearningStep(96, undefined, undefined),

new LearningStep(97, undefined, undefined),

new LearningStep(98, undefined, undefined),

new LearningStep(99, undefined, undefined),

new LearningStep(100, undefined, undefined),

new LearningStep(101, undefined, undefined),

new LearningStep(102, undefined, undefined),

new LearningStep(103, undefined, undefined),

new LearningStep(104, undefined, undefined),

new LearningStep(105, undefined, undefined),

new LearningStep(106, undefined, undefined),

new LearningStep(107, undefined, undefined),

new LearningStep(108, undefined, undefined),

new LearningStep(109, undefined, undefined),

new LearningStep(110, undefined, undefined),
```

```
new LearningStep(
  110,
  "",

  new Question(
    "What is the value of total?",
    [1,0,100],
    ["","",""] ) ),

new LearningStep(
  110,
  "",

  new Question(
    "What is the value of position?",
    [5,10,0],
    ["","",""] ) ),

new LearningStep(111, undefined, undefined),

new LearningStep(112, undefined, undefined),

new LearningStep(113, undefined, undefined),

new LearningStep(114, undefined, undefined),

new LearningStep(
  115,
  "",

  new Question(
    "What did the condition position < numbers.length
         evaluate to?",
    [true,10,-10],
    ["","The answer is a boolean, not a number.","The answer
         is a boolean, not a number."] ) ),

new LearningStep(115, undefined, undefined),

new LearningStep(116, undefined, undefined),

new LearningStep(
  116,
  "Since the word position is long, programmers usually use
       the name i or j for the variable that looks up the
       position in the array. This is a standard practice
       that makes it easier to read code (once you know the
       convention).",
  undefined ),

new LearningStep(
  116,
  "You can just quickly step through this. It works the same
       way as the previous while loop.",
  undefined ),

new LearningStep(117, undefined, undefined),

new LearningStep(118, undefined, undefined),

new LearningStep(119, undefined, undefined),

new LearningStep(120, undefined, undefined),

new LearningStep(121, undefined, undefined),

new LearningStep(122, undefined, undefined),

new LearningStep(123, undefined, undefined),

new LearningStep(124, undefined, undefined),

new LearningStep(125, undefined, undefined),

new LearningStep(126, undefined, undefined),

new LearningStep(127, undefined, undefined),

new LearningStep(128, undefined, undefined),

new LearningStep(129, undefined, undefined),

new LearningStep(130, undefined, undefined),

new LearningStep(131, undefined, undefined),

new LearningStep(132, undefined, undefined),

new LearningStep(133, undefined, undefined),

new LearningStep(134, undefined, undefined),

new LearningStep(135, undefined, undefined),

new LearningStep(136, undefined, undefined),

new LearningStep(137, undefined, undefined),

new LearningStep(138, undefined, undefined),

new LearningStep(139, undefined, undefined),

new LearningStep(140, undefined, undefined),

new LearningStep(141, undefined, undefined),

new LearningStep(142, undefined, undefined),

new LearningStep(143, undefined, undefined),

new LearningStep(144, undefined, undefined),

new LearningStep(145, undefined, undefined),

new LearningStep(146, undefined, undefined),

new LearningStep(147, undefined, undefined),

new LearningStep(148, undefined, undefined),

new LearningStep(149, undefined, undefined),

new LearningStep(150, undefined, undefined),

new LearningStep(151, undefined, undefined),

new LearningStep(152, undefined, undefined),

new LearningStep(153, undefined, undefined),

new LearningStep(154, undefined, undefined),

new LearningStep(155, undefined, undefined),

new LearningStep(156, undefined, undefined),

new LearningStep(157, undefined, undefined),

new LearningStep(158, undefined, undefined),

new LearningStep(159, undefined, undefined),

new LearningStep(160, undefined, undefined),

new LearningStep(161, undefined, undefined),

new LearningStep(162, undefined, undefined),

new LearningStep(163, undefined, undefined),

new LearningStep(164, undefined, undefined),

new LearningStep(165, undefined, undefined),

new LearningStep(166, undefined, undefined),

new LearningStep(167, undefined, undefined),

new LearningStep(168, undefined, undefined),

new LearningStep(169, undefined, undefined),

new LearningStep(170, undefined, undefined),

new LearningStep(171, undefined, undefined),

new LearningStep(172, undefined, undefined),

new LearningStep(
  172,
```

```
  "You can also add up the numbers by **starting with the
       last** and adding until you reach the first. The
       style most people use is to **start from the first
       like the earlier examples though**.",
  undefined ),

new LearningStep(
  172,
  "You can just step through this example, quickly if you
       understand it already, more carefully if it seems odd
        to you (and it is a bit odd).",
  undefined ),

new LearningStep(173, undefined, undefined),

new LearningStep(174, undefined, undefined),

new LearningStep(175, undefined, undefined),

new LearningStep(176, undefined, undefined),

new LearningStep(177, undefined, undefined),

new LearningStep(178, undefined, undefined),

new LearningStep(179, undefined, undefined),

new LearningStep(180, undefined, undefined),

new LearningStep(181, undefined, undefined),

new LearningStep(182, undefined, undefined),

new LearningStep(183, undefined, undefined),

new LearningStep(184, undefined, undefined),

new LearningStep(185, undefined, undefined),

new LearningStep(186, undefined, undefined),

new LearningStep(187, undefined, undefined),

new LearningStep(188, undefined, undefined),

new LearningStep(189, undefined, undefined),

new LearningStep(190, undefined, undefined),

new LearningStep(191, undefined, undefined),

new LearningStep(192, undefined, undefined),

new LearningStep(193, undefined, undefined),

new LearningStep(194, undefined, undefined),

new LearningStep(195, undefined, undefined),

new LearningStep(196, undefined, undefined),

new LearningStep(197, undefined, undefined),

new LearningStep(198, undefined, undefined),

new LearningStep(199, undefined, undefined),

new LearningStep(200, undefined, undefined),

new LearningStep(201, undefined, undefined),

new LearningStep(202, undefined, undefined),

new LearningStep(203, undefined, undefined),

new LearningStep(204, undefined, undefined),

new LearningStep(205, undefined, undefined),

new LearningStep(206, undefined, undefined),

new LearningStep(207, undefined, undefined),

new LearningStep(208, undefined, undefined),

new LearningStep(209, undefined, undefined),

new LearningStep(210, undefined, undefined),

new LearningStep(211, undefined, undefined),

new LearningStep(212, undefined, undefined),

new LearningStep(213, undefined, undefined),

new LearningStep(214, undefined, undefined),

new LearningStep(215, undefined, undefined),

new LearningStep(216, undefined, undefined),

new LearningStep(217, undefined, undefined),

new LearningStep(218, undefined, undefined),

new LearningStep(219, undefined, undefined),

new LearningStep(220, undefined, undefined),

new LearningStep(221, undefined, undefined),

new LearningStep(222, undefined, undefined),

new LearningStep(223, undefined, undefined),

new LearningStep(224, undefined, undefined),

new LearningStep(225, undefined, undefined),

new LearningStep(226, undefined, undefined),

new LearningStep(
  227,
  " ",
  undefined ),

new LearningStep(228, undefined, undefined),

new LearningStep(229, undefined, undefined),

new LearningStep(
  230,
  " Great! Let's go forward!",
  undefined )]
                    ) /* end LearningSteps */
                    ) /* end Action */,
new Action (
                    new Program(`  var a = [1,10];
var total_of_a = 0;

var i = 0;
while ( i < a.length ){
  total_of_a = total_of_a + a[i];
  i = i + 1;
}

var a = [1,10];
var total_of_a = 0;

for (var i = 0; i < a.length ; i = i + 1 ){
  total_of_a = total_of_a + a[i];
}


var condition = true;
while ( condition ) {
    // more code usually here
    var x = 1;
    condition = false;
}

for (var condition = true; condition; condition = false){
    // more code usually here
    var x = 1;
}

var how_many_times_for = 0;
for (var cond_for = true; cond_for; cond_for = false){
    how_many_times_for =
        how_many_times_for + 1;
```

```
}
            `
            ) /* end Program */,
                    new LearningSteps(
        /* Learning Steps */
        [
  new LearningStep(
    0,
    "## For loops - a common pattern\n \nThere is a common
         pattern when writing while\nloops. You want to do
         something for each\nvalue that is in an **array**.\n\
         n
    For loops are a short way of writing loops\nthat have that
         kind of pattern.\n\n
    This while loop and for loop\nare basically equivalent.\n\n
    For loops often are used with arrays\nbecause they are
         shorter.\nCompare the while and for loops",
    undefined ),

  new LearningStep(0, undefined, undefined),

  new LearningStep(1, undefined, undefined),

  new LearningStep(2, undefined, undefined),

  new LearningStep(3, undefined, undefined),

  new LearningStep(4, undefined, undefined),

  new LearningStep(5, undefined, undefined),

  new LearningStep(6, undefined, undefined),

  new LearningStep(7, undefined, undefined),

  new LearningStep(8, undefined, undefined),

  new LearningStep(9, undefined, undefined),

  new LearningStep(10, undefined, undefined),

  new LearningStep(11, undefined, undefined),

  new LearningStep(12, undefined, undefined),

  new LearningStep(13, undefined, undefined),

  new LearningStep(14, undefined, undefined),

  new LearningStep(15, undefined, undefined),

  new LearningStep(16, undefined, undefined),

  new LearningStep(17, undefined, undefined),

  new LearningStep(18, undefined, undefined),

  new LearningStep(19, undefined, undefined),

  new LearningStep(20, undefined, undefined),

  new LearningStep(21, undefined, undefined),

  new LearningStep(22, undefined, undefined),

  new LearningStep(23, undefined, undefined),

  new LearningStep(24, undefined, undefined),

  new LearningStep(25, undefined, undefined),

  new LearningStep(26, undefined, undefined),

  new LearningStep(27, undefined, undefined),

  new LearningStep(28, undefined, undefined),

  new LearningStep(29, undefined, undefined),

  new LearningStep(30, undefined, undefined),

  new LearningStep(31, undefined, undefined),

  new LearningStep(32, undefined, undefined),

  new LearningStep(33, undefined, undefined),

  new LearningStep(34, undefined, undefined),

  new LearningStep(35, undefined, undefined),

  new LearningStep(36, undefined, undefined),

  new LearningStep(37, undefined, undefined),

  new LearningStep(38, undefined, undefined),

  new LearningStep(39, undefined, undefined),

  new LearningStep(40, undefined, undefined),

  new LearningStep(41, undefined, undefined),

  new LearningStep(42, undefined, undefined),

  new LearningStep(43, undefined, undefined),

  new LearningStep(44, undefined, undefined),

  new LearningStep(45, undefined, undefined),

  new LearningStep(46, undefined, undefined),

  new LearningStep(47, undefined, undefined),

  new LearningStep(48, undefined, undefined),

  new LearningStep(49, undefined, undefined),

  new LearningStep(50, undefined, undefined),

  new LearningStep(51, undefined, undefined),

  new LearningStep(52, undefined, undefined),

  new LearningStep(53, undefined, undefined),

  new LearningStep(54, undefined, undefined),

  new LearningStep(55, undefined, undefined),

  new LearningStep(56, undefined, undefined),

  new LearningStep(57, undefined, undefined),

  new LearningStep(58, undefined, undefined),

  new LearningStep(59, undefined, undefined),

  new LearningStep(60, undefined, undefined),

  new LearningStep(
    60,
    "The syntax of a **for** loop is:\n```\nfor ( init;
         condition; incr){\n  code for the body goes here\n}\n
         ```\n\n
    init is a place to put code that creates variables to use
          in the loop.\n\n
    condition is like the condition for a while loop.\n\n
    incr is a place to put code that is executed after the
         body is executed **for each time** the body is
         executed. Usually this increments a variable used to
         look up a position in the array.\n\n
    The **for** loop is named to represent this **for each time
         ** behavior.\n",
    undefined),

  new LearningStep(61, undefined, undefined),

  new LearningStep(62, undefined, undefined),

  new LearningStep(63, undefined, undefined),

  new LearningStep(64, undefined, undefined),

  new LearningStep(65, undefined, undefined),

  new LearningStep(66, undefined, undefined),
```
```

```
new LearningStep(67, undefined, undefined),

new LearningStep(68, undefined, undefined),

new LearningStep(69, undefined, undefined),

new LearningStep(70, undefined, undefined),

new LearningStep(71, undefined, undefined),

new LearningStep(72, undefined, undefined),

new LearningStep(73, undefined, undefined),

new LearningStep(
  73,
  "Step through this part slowly and read the instruction
       descriptions.\n",
  undefined ),

new LearningStep(74, undefined, undefined),

new LearningStep(75, undefined, undefined),

new LearningStep(76, undefined, undefined),

new LearningStep(77, undefined, undefined),

new LearningStep(78, undefined, undefined),

new LearningStep(
  78,
  "The syntax of a **for** loop is:\n```\nfor ( init;
       condition; incr){\n  code for the body goes here\n}\n
       ```\nWe just executed the *init* part. It only gets
       executed once.\n*init* is a place to put code that
       creates variables to use in the loop.\n",
  undefined),

new LearningStep(
  78,
  " ",
  undefined),

new LearningStep(79, undefined, undefined),

new LearningStep(80, undefined, undefined),

new LearningStep(81, undefined, undefined),

new LearningStep(82, undefined, undefined),

new LearningStep(83, undefined, undefined),

new LearningStep(84, undefined, undefined),

new LearningStep(85, undefined, undefined),

new LearningStep(86, undefined, undefined),

new LearningStep(87, undefined, undefined),

new LearningStep(88, undefined, undefined),

new LearningStep(89, undefined, undefined),

new LearningStep(90, undefined, undefined),

new LearningStep(91, undefined, undefined),

new LearningStep(
  91,
  " The syntax of a **for** loop is:\n```\nfor (init;
       condition; incr){\n code for the body goes here\n}\n
       ```\nNow, instead of checking the condition right
       away after the body, we do the *incr* steps.\n\n
  *incr* is a place to put code that is executed after the
       body is executed **for each time** the body is
       executed.",
  undefined ),

new LearningStep(92, undefined, undefined),

new LearningStep(93, undefined, undefined),

new LearningStep(94, undefined, undefined),

new LearningStep(95, undefined, undefined),

new LearningStep(96, undefined, undefined),

new LearningStep(
  97,
  "Now that we're done with the **increment** section of the
       **for**, we check the condition again.",
  undefined ),

new LearningStep(98, undefined, undefined),

new LearningStep(99, undefined, undefined),

new LearningStep(100, undefined, undefined),

new LearningStep(101, undefined, undefined),

new LearningStep(102, undefined, undefined),

new LearningStep(103, undefined, undefined),

new LearningStep(104, undefined, undefined),

new LearningStep(105, undefined, undefined),

new LearningStep(106, undefined, undefined),

new LearningStep(107, undefined, undefined),

new LearningStep(108, undefined, undefined),

new LearningStep(109, undefined, undefined),

new LearningStep(110, undefined, undefined),

new LearningStep(111, undefined, undefined),

new LearningStep(112, undefined, undefined),

new LearningStep(
  112,
  "Now, instead of checking the condition right away after
       the body, we do the incr steps.",
  undefined ),

new LearningStep(113, undefined, undefined),

new LearningStep(114, undefined, undefined),

new LearningStep(115, undefined, undefined),

new LearningStep(116, undefined, undefined),

new LearningStep(117, undefined, undefined),

new LearningStep(118, undefined, undefined),

new LearningStep(119, undefined, undefined),

new LearningStep(120, undefined, undefined),

new LearningStep(121, undefined, undefined),

new LearningStep(122, undefined, undefined),

new LearningStep(123, undefined, undefined),

new LearningStep(124, undefined, undefined),

new LearningStep(125, undefined, undefined),

new LearningStep(126, undefined, undefined),

new LearningStep(
  126,
  "",

  new Question(
    "What is total_of_a?",
    [5,0,100],
    ["","",""] ) ),
```

```
new LearningStep(
  126,
  "## Why for loops\n\n
  The **for** loop makes writing loops simpler when they
       follow this pattern:\n\n
  There is a variable that\ngets an initial value. Then that
       variable\nis used as part of the condition, and there
       \nis some statement at the end of the body that\
       nchanges that variable.\n\n
  Compared to a **while** loop, the **for** loop makes that
       pattern **more explicit via its syntax**.",
  undefined ),

new LearningStep(
  126,
  "Let's go through some simpler examples. Pay attention to
       the **order of execution** for the code in the init (
       initialization), condition, and incr (increment)
       parts.\n\n
  ```\nfor ( init; condition; incr) {\n   body code to repeat
        goes here\n}\n```",
  undefined),

new LearningStep(127, undefined, undefined),

new LearningStep(128, undefined, undefined),

new LearningStep(129, undefined, undefined),

new LearningStep(130, undefined, undefined),

new LearningStep(131, undefined, undefined),

new LearningStep(132, undefined, undefined),

new LearningStep(133, undefined, undefined),

new LearningStep(134, undefined, undefined),

new LearningStep(135, undefined, undefined),

new LearningStep(136, undefined, undefined),

new LearningStep(137, undefined, undefined),

new LearningStep(138, undefined, undefined),

new LearningStep(139, undefined, undefined),

new LearningStep(140, undefined, undefined),

new LearningStep(141, undefined, undefined),

new LearningStep(142, undefined, undefined),

new LearningStep(143, undefined, undefined),

new LearningStep(144, undefined, undefined),

new LearningStep(145, undefined, undefined),

new LearningStep(146, undefined, undefined),

new LearningStep(147, undefined, undefined),

new LearningStep(148, undefined, undefined),

new LearningStep(149, undefined, undefined),

new LearningStep(150, undefined, undefined),

new LearningStep(151, undefined, undefined),

new LearningStep(152, undefined, undefined),

new LearningStep(153, undefined, undefined),

new LearningStep(154, undefined, undefined),

new LearningStep(155, undefined, undefined),

new LearningStep(156, undefined, undefined),

new LearningStep(157, undefined, undefined),

new LearningStep(158, undefined, undefined),

new LearningStep(159, undefined, undefined),

new LearningStep(160, undefined, undefined),

new LearningStep(161, undefined, undefined),

new LearningStep(162, undefined, undefined),

new LearningStep(163, undefined, undefined),

new LearningStep(164, undefined, undefined),

new LearningStep(165, undefined, undefined),

new LearningStep(166, undefined, undefined),

new LearningStep(167, undefined, undefined),

new LearningStep(
  167,
  "Now, try mentally executing this for loop. What does it do
       ?",
  undefined),

new LearningStep(168, undefined, undefined),

new LearningStep(169, undefined, undefined),

new LearningStep(170, undefined, undefined),

new LearningStep(171, undefined, undefined),

new LearningStep(172, undefined, undefined),

new LearningStep(173, undefined, undefined),

new LearningStep(174, undefined, undefined),

new LearningStep(175, undefined, undefined),

new LearningStep(176, undefined, undefined),

new LearningStep(177, undefined, undefined),

new LearningStep(178, undefined, undefined),

new LearningStep(179, undefined, undefined),

new LearningStep(180, undefined, undefined),

new LearningStep(181, undefined, undefined),

new LearningStep(182, undefined, undefined),

new LearningStep(183, undefined, undefined),

new LearningStep(184, undefined, undefined),

new LearningStep(185, undefined, undefined),

new LearningStep(186, undefined, undefined),

new LearningStep(187, undefined, undefined),

new LearningStep(188, undefined, undefined),

new LearningStep(189, undefined, undefined),

new LearningStep(190, undefined, undefined),

new LearningStep(191, undefined, undefined),

new LearningStep(192, undefined, undefined),

new LearningStep(193, undefined, undefined),

new LearningStep(
  194,
  "",

  new Question(
    "What is the value of how_many_times_for?",
```
```

```
      [5,100,10],
      ["","",""] ) ),

  new LearningStep(
    194,
    "Great! Onwards!",
    undefined )]
                    ) /* end LearningSteps */
                    ) /* end Action */,
new Action (
                    new Program(` var total = 0;

do {
 total = total + 10;
} while (total < 30);

total;


var c = 0;

do {
 c = c - 2;
}
while ( c > 2 );

var d = 100;
var e = 10;
var f = 0;

do {
  d = d - e;
  f = f - 1;
} while (f > 0);

d = 100;
e = 10;
f = 3;
do {
  d = d - e;
  f = f - 1;
} while (f > 0);

            `
            ) /* end Program */,
                    new LearningSteps(
        /* Learning Steps */
        [
  new LearningStep(
    0,
    "## Do While Loops\n\n
    Sometimes you want to execute\nsome part of a program **
          once,\nno matter what,** then *maybe* do that again.\
          n\n
    **Do While** loops have been made\nfor this purpose.\n\n
    They are a lot like **while** loops.\n\n
    Let's start with\nsome examples to see how\ndo while loops
          work.\n\n
    This first do while\nadds 10 to total\nwhile total is less
          than\n30.",
    undefined ),

  new LearningStep(0, undefined, undefined),

  new LearningStep(1, undefined, undefined),

  new LearningStep(2, undefined, undefined),

  new LearningStep(3, undefined, undefined),

  new LearningStep(4, undefined, undefined),

  new LearningStep(5, undefined, undefined),

  new LearningStep(6, undefined, undefined),

  new LearningStep(7, undefined, undefined),

  new LearningStep(8, undefined, undefined),

  new LearningStep(9, undefined, undefined),

  new LearningStep(10, undefined, undefined),

  new LearningStep(11, undefined, undefined),

  new LearningStep(12, undefined, undefined),

  new LearningStep(13, undefined, undefined),

  new LearningStep(14, undefined, undefined),

  new LearningStep(15, undefined, undefined),

  new LearningStep(16, undefined, undefined),

  new LearningStep(17, undefined, undefined),

  new LearningStep(18, undefined, undefined),

  new LearningStep(19, undefined, undefined),

  new LearningStep(20, undefined, undefined),

  new LearningStep(21, undefined, undefined),

  new LearningStep(22, undefined, undefined),

  new LearningStep(23, undefined, undefined),

  new LearningStep(24, undefined, undefined),

  new LearningStep(25, undefined, undefined),

  new LearningStep(26, undefined, undefined),

  new LearningStep(27, undefined, undefined),

  new LearningStep(28, undefined, undefined),

  new LearningStep(29, undefined, undefined),

  new LearningStep(30, undefined, undefined),

  new LearningStep(31, undefined, undefined),

  new LearningStep(32, undefined, undefined),

  new LearningStep(33, undefined, undefined),

  new LearningStep(34, undefined, undefined),

  new LearningStep(35, undefined, undefined),

  new LearningStep(36, undefined, undefined),

new LearningStep(
  36,
  undefined,

  new Question(
    "What value is on the stack?",
    [0,5,1],
    ["","",""] ) ),

  new LearningStep(37, undefined, undefined),

  new LearningStep(38, undefined, undefined),

  new LearningStep(39, undefined, undefined),

  new LearningStep(40, undefined, undefined),

  new LearningStep(41, undefined, undefined),

  new LearningStep(42, undefined, undefined),

  new LearningStep(43, undefined, undefined),

  new LearningStep(44, undefined, undefined),

  new LearningStep(45, undefined, undefined),

  new LearningStep(46, undefined, undefined),

  new LearningStep(47, undefined, undefined),

  new LearningStep(48, undefined, undefined),

  new LearningStep(49, undefined, undefined),
```

```
new LearningStep(50, undefined, undefined),

new LearningStep(51, undefined, undefined),

new LearningStep(52, undefined, undefined),

new LearningStep(
  52,
  " ",

  new Question(
    "What's the value of c?",
    [5,100,10],
    ["Do-while executes the do statement once no matter what,
          then checks the condition.","Do-while executes the
          do statement once no matter what, then checks the
         condition.","Do-while executes the do statement
         once no matter what, then checks the condition."] )
          ),

new LearningStep(53, undefined, undefined),

new LearningStep(54, undefined, undefined),

new LearningStep(55, undefined, undefined),

new LearningStep(56, undefined, undefined),

new LearningStep(57, undefined, undefined),

new LearningStep(58, undefined, undefined),

new LearningStep(59, undefined, undefined),

new LearningStep(60, undefined, undefined),

new LearningStep(61, undefined, undefined),

new LearningStep(62, undefined, undefined),

new LearningStep(63, undefined, undefined),

new LearningStep(64, undefined, undefined),

new LearningStep(65, undefined, undefined),

new LearningStep(66, undefined, undefined),

new LearningStep(67, undefined, undefined),

new LearningStep(68, undefined, undefined),

new LearningStep(69, undefined, undefined),

new LearningStep(70, undefined, undefined),

new LearningStep(71, undefined, undefined),

new LearningStep(72, undefined, undefined),

new LearningStep(
  72,
  undefined,

  new Question(
    "What value is on the stack?",
    [1,10,5],
    ["","",""] ) ),

new LearningStep(73, undefined, undefined),

new LearningStep(74, undefined, undefined),

new LearningStep(75, undefined, undefined),

new LearningStep(76, undefined, undefined),

new LearningStep(77, undefined, undefined),

new LearningStep(
  77,
  undefined,

  new Question(
    "What value is on the stack?",
    [100,5,1],
    ["","",""] ) ),

new LearningStep(78, undefined, undefined),

new LearningStep(79, undefined, undefined),

new LearningStep(80, undefined, undefined),

new LearningStep(81, undefined, undefined),

new LearningStep(82, undefined, undefined),

new LearningStep(83, undefined, undefined),

new LearningStep(84, undefined, undefined),

new LearningStep(85, undefined, undefined),

new LearningStep(86, undefined, undefined),

new LearningStep(87, undefined, undefined),

new LearningStep(88, undefined, undefined),

new LearningStep(89, undefined, undefined),

new LearningStep(90, undefined, undefined),

new LearningStep(91, undefined, undefined),

new LearningStep(92, undefined, undefined),

new LearningStep(93, undefined, undefined),

new LearningStep(94, undefined, undefined),

new LearningStep(95, undefined, undefined),

new LearningStep(96, undefined, undefined),

new LearningStep(97, undefined, undefined),

new LearningStep(98, undefined, undefined),

new LearningStep(99, undefined, undefined),

new LearningStep(100, undefined, undefined),

new LearningStep(101, undefined, undefined),

new LearningStep(
  101,
  undefined,

  new Question(
    "What value is on the stack?",
    [5,0,10],
    ["","",""] ) ),

new LearningStep(102, undefined, undefined),

new LearningStep(103, undefined, undefined),

new LearningStep(104, undefined, undefined),

new LearningStep(105, undefined, undefined),

new LearningStep(106, undefined, undefined),

new LearningStep(107, undefined, undefined),

new LearningStep(108, undefined, undefined),

new LearningStep(109, undefined, undefined),

new LearningStep(110, undefined, undefined),

new LearningStep(111, undefined, undefined),

new LearningStep(112, undefined, undefined),

new LearningStep(113, undefined, undefined),
```

```
  new LearningStep(114, undefined, undefined),

  new LearningStep(115, undefined, undefined),

  new LearningStep(116, undefined, undefined),

  new LearningStep(
    116,
    undefined,

    new Question(
      "What value is on the stack?",
      [100,5,10],
      ["","",""] ) ),

  new LearningStep(117, undefined, undefined),

  new LearningStep(118, undefined, undefined),

  new LearningStep(119, undefined, undefined),

  new LearningStep(120, undefined, undefined),

  new LearningStep(121, undefined, undefined),

  new LearningStep(122, undefined, undefined),

  new LearningStep(123, undefined, undefined),

  new LearningStep(124, undefined, undefined),

  new LearningStep(125, undefined, undefined),

  new LearningStep(126, undefined, undefined),

  new LearningStep(127, undefined, undefined),

  new LearningStep(128, undefined, undefined),

  new LearningStep(129, undefined, undefined),

  new LearningStep(130, undefined, undefined),

  new LearningStep(131, undefined, undefined),

  new LearningStep(
    131,
    undefined,

    new Question(
      "What value is on the stack?",
      [10,1,5],
      ["","",""] ) ),

  new LearningStep(132, undefined, undefined),

  new LearningStep(133, undefined, undefined),

  new LearningStep(134, undefined, undefined),

  new LearningStep(135, undefined, undefined)]
                    ) /* end LearningSteps */
                    ) /* end Action */,
new Action (
                    new Program(`

var rooms = [ [12, 10, 8],
              [20, 12, 8],
              [13, 15, 8],
              [8, 12, 8],
              [20, 30, 8]];

var total_volume = 0;
for (var i = 0; i < rooms.length; i = i + 1)
{
  room = rooms[i];
  room_volume = room[0] * room[1] * room[2];
  total_volume = total_volume + room_volume;
}

var ACs = [ [300, 9000, 1],
            [500, 11000, 2],
            [450, 12000, 3]];

var can_work_ACs = [];

for (var i = 0; i < ACs.length; i = i + 1)
{
  var this_ac = ACs[i];
  if (total_volume < this_ac[1])
  {
     var l = can_work_ACs.length;
     can_work_ACs[l] = this_ac;
  }
}

var best_price = 1000000000;
var best_ac_number = -1;

for (var i = 0; i < can_work_ACs.length; i = i + 1)
{
  var this_ac = can_work_ACs[i];
  this_acs_price = this_ac[0];
  if ( this_acs_price < best_price )
  {
     best_price = this_acs_price;
     this_acs_number = this_ac[2];
     best_ac_number = this_acs_number;
  }

}

best_ac_number;


            `
            ) /* end Program */,
                    new LearningSteps(
        /* Learning Steps */
        [
  new LearningStep(
    0,
    "## Using loops\n\n
    Now we want to recommend\nthe cheapest air conditioner\
        nthat can handle some house's volume.\n\n
    You can go to the Ifs section\nthen click on the last
        program\nto see our prior version of this.\nWe'll
        show some of it here.\n\n
    We'll extend it now\nextend it using a loop.\n\n
    First we start with calculating the volume we need to cool
        .\n",
    undefined ),

  new LearningStep(
    0,
    "You can step through this program quickly, it is just an
        example.",
    undefined ),

  new LearningStep(0, undefined, undefined),

  new LearningStep(1, undefined, undefined),

  new LearningStep(2, undefined, undefined),

  new LearningStep(3, undefined, undefined),

  new LearningStep(4, undefined, undefined),

  new LearningStep(5, undefined, undefined),

  new LearningStep(6, undefined, undefined),

  new LearningStep(7, undefined, undefined),

  new LearningStep(8, undefined, undefined),

  new LearningStep(9, undefined, undefined),

  new LearningStep(10, undefined, undefined),

  new LearningStep(11, undefined, undefined),

  new LearningStep(12, undefined, undefined),

  new LearningStep(13, undefined, undefined),

  new LearningStep(14, undefined, undefined),
```

```
new LearningStep(15, undefined, undefined),
new LearningStep(16, undefined, undefined),
new LearningStep(17, undefined, undefined),
new LearningStep(18, undefined, undefined),
new LearningStep(19, undefined, undefined),
new LearningStep(20, undefined, undefined),
new LearningStep(21, undefined, undefined),
new LearningStep(22, undefined, undefined),
new LearningStep(23, undefined, undefined),
new LearningStep(24, undefined, undefined),
new LearningStep(25, undefined, undefined),
new LearningStep(26, undefined, undefined),
new LearningStep(27, undefined, undefined),
new LearningStep(28, undefined, undefined),
new LearningStep(29, undefined, undefined),
new LearningStep(30, undefined, undefined),
new LearningStep(31, undefined, undefined),
new LearningStep(32, undefined, undefined),
new LearningStep(33, undefined, undefined),
new LearningStep(34, undefined, undefined),
new LearningStep(35, undefined, undefined),
new LearningStep(36, undefined, undefined),
new LearningStep(37, undefined, undefined),
new LearningStep(38, undefined, undefined),
new LearningStep(39, undefined, undefined),
new LearningStep(40, undefined, undefined),
new LearningStep(41, undefined, undefined),
new LearningStep(42, undefined, undefined),
new LearningStep(43, undefined, undefined),
new LearningStep(44, undefined, undefined),
new LearningStep(45, undefined, undefined),
new LearningStep(46, undefined, undefined),
new LearningStep(47, undefined, undefined),
new LearningStep(48, undefined, undefined),
new LearningStep(49, undefined, undefined),
new LearningStep(50, undefined, undefined),
new LearningStep(51, undefined, undefined),
new LearningStep(52, undefined, undefined),
new LearningStep(53, undefined, undefined),
new LearningStep(54, undefined, undefined),
new LearningStep(55, undefined, undefined),
new LearningStep(56, undefined, undefined),
new LearningStep(57, undefined, undefined),
new LearningStep(58, undefined, undefined),
new LearningStep(59, undefined, undefined),
new LearningStep(60, undefined, undefined),
new LearningStep(61, undefined, undefined),
new LearningStep(62, undefined, undefined),
new LearningStep(63, undefined, undefined),
new LearningStep(64, undefined, undefined),
new LearningStep(65, undefined, undefined),
new LearningStep(66, undefined, undefined),
new LearningStep(67, undefined, undefined),
new LearningStep(68, undefined, undefined),
new LearningStep(69, undefined, undefined),
new LearningStep(70, undefined, undefined),
new LearningStep(71, undefined, undefined),
new LearningStep(72, undefined, undefined),
new LearningStep(73, undefined, undefined),
new LearningStep(74, undefined, undefined),
new LearningStep(75, undefined, undefined),
new LearningStep(76, undefined, undefined),
new LearningStep(77, undefined, undefined),
new LearningStep(78, undefined, undefined),
new LearningStep(79, undefined, undefined),
new LearningStep(80, undefined, undefined),
new LearningStep(81, undefined, undefined),
new LearningStep(82, undefined, undefined),
new LearningStep(83, undefined, undefined),
new LearningStep(84, undefined, undefined),
new LearningStep(85, undefined, undefined),
new LearningStep(86, undefined, undefined),
new LearningStep(87, undefined, undefined),
new LearningStep(88, undefined, undefined),
new LearningStep(89, undefined, undefined),
new LearningStep(90, undefined, undefined),
new LearningStep(91, undefined, undefined),
new LearningStep(92, undefined, undefined),
new LearningStep(93, undefined, undefined),
new LearningStep(94, undefined, undefined),
new LearningStep(95, undefined, undefined),
new LearningStep(96, undefined, undefined),
new LearningStep(97, undefined, undefined),
new LearningStep(98, undefined, undefined),
new LearningStep(99, undefined, undefined),
```

```
new LearningStep(100, undefined, undefined),
new LearningStep(101, undefined, undefined),
new LearningStep(102, undefined, undefined),
new LearningStep(103, undefined, undefined),
new LearningStep(104, undefined, undefined),
new LearningStep(105, undefined, undefined),
new LearningStep(106, undefined, undefined),
new LearningStep(107, undefined, undefined),
new LearningStep(108, undefined, undefined),
new LearningStep(109, undefined, undefined),
new LearningStep(110, undefined, undefined),
new LearningStep(111, undefined, undefined),
new LearningStep(112, undefined, undefined),
new LearningStep(113, undefined, undefined),
new LearningStep(114, undefined, undefined),
new LearningStep(115, undefined, undefined),
new LearningStep(116, undefined, undefined),
new LearningStep(117, undefined, undefined),
new LearningStep(118, undefined, undefined),
new LearningStep(119, undefined, undefined),
new LearningStep(120, undefined, undefined),
new LearningStep(121, undefined, undefined),
new LearningStep(122, undefined, undefined),
new LearningStep(123, undefined, undefined),
new LearningStep(124, undefined, undefined),
new LearningStep(125, undefined, undefined),
new LearningStep(126, undefined, undefined),
new LearningStep(127, undefined, undefined),
new LearningStep(128, undefined, undefined),
new LearningStep(129, undefined, undefined),
new LearningStep(130, undefined, undefined),
new LearningStep(131, undefined, undefined),
new LearningStep(132, undefined, undefined),
new LearningStep(133, undefined, undefined),
new LearningStep(134, undefined, undefined),
new LearningStep(135, undefined, undefined),
new LearningStep(136, undefined, undefined),
new LearningStep(137, undefined, undefined),
new LearningStep(138, undefined, undefined),
new LearningStep(139, undefined, undefined),
new LearningStep(140, undefined, undefined),
new LearningStep(141, undefined, undefined),
new LearningStep(142, undefined, undefined),
new LearningStep(143, undefined, undefined),
new LearningStep(144, undefined, undefined),
new LearningStep(145, undefined, undefined),
new LearningStep(146, undefined, undefined),
new LearningStep(147, undefined, undefined),
new LearningStep(148, undefined, undefined),
new LearningStep(149, undefined, undefined),
new LearningStep(150, undefined, undefined),
new LearningStep(151, undefined, undefined),
new LearningStep(152, undefined, undefined),
new LearningStep(153, undefined, undefined),
new LearningStep(154, undefined, undefined),
new LearningStep(155, undefined, undefined),
new LearningStep(156, undefined, undefined),
new LearningStep(157, undefined, undefined),
new LearningStep(158, undefined, undefined),
new LearningStep(159, undefined, undefined),
new LearningStep(160, undefined, undefined),
new LearningStep(161, undefined, undefined),
new LearningStep(162, undefined, undefined),
new LearningStep(163, undefined, undefined),
new LearningStep(164, undefined, undefined),
new LearningStep(165, undefined, undefined),
new LearningStep(166, undefined, undefined),
new LearningStep(167, undefined, undefined),
new LearningStep(168, undefined, undefined),
new LearningStep(169, undefined, undefined),
new LearningStep(170, undefined, undefined),
new LearningStep(171, undefined, undefined),
new LearningStep(172, undefined, undefined),
new LearningStep(173, undefined, undefined),
new LearningStep(174, undefined, undefined),
new LearningStep(175, undefined, undefined),
new LearningStep(176, undefined, undefined),
new LearningStep(177, undefined, undefined),
new LearningStep(178, undefined, undefined),
new LearningStep(179, undefined, undefined),
new LearningStep(180, undefined, undefined),
new LearningStep(181, undefined, undefined),
new LearningStep(182, undefined, undefined),
new LearningStep(183, undefined, undefined),
new LearningStep(184, undefined, undefined),
```

```
new LearningStep(185, undefined, undefined),
new LearningStep(186, undefined, undefined),
new LearningStep(187, undefined, undefined),
new LearningStep(188, undefined, undefined),
new LearningStep(189, undefined, undefined),
new LearningStep(190, undefined, undefined),
new LearningStep(191, undefined, undefined),
new LearningStep(192, undefined, undefined),
new LearningStep(193, undefined, undefined),
new LearningStep(194, undefined, undefined),
new LearningStep(195, undefined, undefined),
new LearningStep(196, undefined, undefined),
new LearningStep(197, undefined, undefined),
new LearningStep(198, undefined, undefined),
new LearningStep(199, undefined, undefined),
new LearningStep(200, undefined, undefined),
new LearningStep(201, undefined, undefined),
new LearningStep(202, undefined, undefined),
new LearningStep(203, undefined, undefined),
new LearningStep(204, undefined, undefined),
new LearningStep(205, undefined, undefined),
new LearningStep(206, undefined, undefined),
new LearningStep(207, undefined, undefined),
new LearningStep(208, undefined, undefined),
new LearningStep(209, undefined, undefined),
new LearningStep(210, undefined, undefined),
new LearningStep(211, undefined, undefined),
new LearningStep(212, undefined, undefined),
new LearningStep(213, undefined, undefined),
new LearningStep(214, undefined, undefined),
new LearningStep(215, undefined, undefined),
new LearningStep(216, undefined, undefined),
new LearningStep(217, undefined, undefined),
new LearningStep(218, undefined, undefined),
new LearningStep(219, undefined, undefined),
new LearningStep(220, undefined, undefined),
new LearningStep(221, undefined, undefined),
new LearningStep(222, undefined, undefined),
new LearningStep(223, undefined, undefined),
new LearningStep(224, undefined, undefined),
new LearningStep(225, undefined, undefined),
new LearningStep(226, undefined, undefined),
new LearningStep(227, undefined, undefined),
new LearningStep(228, undefined, undefined),
new LearningStep(229, undefined, undefined),
new LearningStep(230, undefined, undefined),
new LearningStep(231, undefined, undefined),
new LearningStep(232, undefined, undefined),
new LearningStep(
  232,
  "We'll store the air conditioner's price, then maximum
       volume it can serve, then a number so we know which
       air conditioner we're recommending. This can be like
       a unique number for that model.",
  undefined ),
new LearningStep(233, undefined, undefined),
new LearningStep(234, undefined, undefined),
new LearningStep(235, undefined, undefined),
new LearningStep(236, undefined, undefined),
new LearningStep(237, undefined, undefined),
new LearningStep(238, undefined, undefined),
new LearningStep(239, undefined, undefined),
new LearningStep(240, undefined, undefined),
new LearningStep(241, undefined, undefined),
new LearningStep(242, undefined, undefined),
new LearningStep(243, undefined, undefined),
new LearningStep(244, undefined, undefined),
new LearningStep(245, undefined, undefined),
new LearningStep(246, undefined, undefined),
new LearningStep(247, undefined, undefined),
new LearningStep(248, undefined, undefined),
new LearningStep(249, undefined, undefined),
new LearningStep(250, undefined, undefined),
new LearningStep(
  250,
  "You can imagine having many more options for air
       conditioners.",
  undefined ),
new LearningStep(251, undefined, undefined),
new LearningStep(252, undefined, undefined),
new LearningStep(253, undefined, undefined),
new LearningStep(
  254,
  "Now we'll use a loop to look through all the air
       conditioners to find the ones that can serve the
       house's volume. We'll add them to an array.",
  undefined ),
new LearningStep(
  254,
  "Now we'll use a loop to look through all the air
       conditioners to find the ones that can serve the
       house's volume. We'll add them to an array.",
  undefined ),
new LearningStep(255, undefined, undefined),
new LearningStep(256, undefined, undefined),
```

```
new LearningStep(257, undefined, undefined),
new LearningStep(258, undefined, undefined),
new LearningStep(259, undefined, undefined),
new LearningStep(260, undefined, undefined),
new LearningStep(261, undefined, undefined),
new LearningStep(262, undefined, undefined),
new LearningStep(263, undefined, undefined),
new LearningStep(264, undefined, undefined),
new LearningStep(265, undefined, undefined),
new LearningStep(266, undefined, undefined),
new LearningStep(267, undefined, undefined),
new LearningStep(268, undefined, undefined),
new LearningStep(269, undefined, undefined),
new LearningStep(270, undefined, undefined),
new LearningStep(271, undefined, undefined),
new LearningStep(272, undefined, undefined),
new LearningStep(273, undefined, undefined),
new LearningStep(274, undefined, undefined),
new LearningStep(275, undefined, undefined),
new LearningStep(276, undefined, undefined),
new LearningStep(277, undefined, undefined),
new LearningStep(278, undefined, undefined),
new LearningStep(279, undefined, undefined),
new LearningStep(280, undefined, undefined),
new LearningStep(281, undefined, undefined),
new LearningStep(282, undefined, undefined),
new LearningStep(283, undefined, undefined),
new LearningStep(284, undefined, undefined),
new LearningStep(285, undefined, undefined),
new LearningStep(286, undefined, undefined),
new LearningStep(287, undefined, undefined),
new LearningStep(288, undefined, undefined),
new LearningStep(289, undefined, undefined),
new LearningStep(290, undefined, undefined),
new LearningStep(291, undefined, undefined),
new LearningStep(292, undefined, undefined),
new LearningStep(293, undefined, undefined),
new LearningStep(294, undefined, undefined),
new LearningStep(295, undefined, undefined),
new LearningStep(296, undefined, undefined),
new LearningStep(297, undefined, undefined),
new LearningStep(298, undefined, undefined),
new LearningStep(299, undefined, undefined),
new LearningStep(300, undefined, undefined),
new LearningStep(301, undefined, undefined),
new LearningStep(302, undefined, undefined),
new LearningStep(303, undefined, undefined),
new LearningStep(304, undefined, undefined),
new LearningStep(305, undefined, undefined),
new LearningStep(306, undefined, undefined),
new LearningStep(307, undefined, undefined),
new LearningStep(308, undefined, undefined),
new LearningStep(309, undefined, undefined),
new LearningStep(310, undefined, undefined),
new LearningStep(311, undefined, undefined),
new LearningStep(312, undefined, undefined),
new LearningStep(313, undefined, undefined),
new LearningStep(314, undefined, undefined),
new LearningStep(315, undefined, undefined),
new LearningStep(316, undefined, undefined),
new LearningStep(317, undefined, undefined),
new LearningStep(318, undefined, undefined),
new LearningStep(319, undefined, undefined),
new LearningStep(320, undefined, undefined),
new LearningStep(321, undefined, undefined),
new LearningStep(322, undefined, undefined),
new LearningStep(323, undefined, undefined),
new LearningStep(324, undefined, undefined),
new LearningStep(325, undefined, undefined),
new LearningStep(326, undefined, undefined),
new LearningStep(327, undefined, undefined),
new LearningStep(328, undefined, undefined),
new LearningStep(329, undefined, undefined),
new LearningStep(330, undefined, undefined),
new LearningStep(331, undefined, undefined),
new LearningStep(332, undefined, undefined),
new LearningStep(333, undefined, undefined),
new LearningStep(334, undefined, undefined),
new LearningStep(335, undefined, undefined),
new LearningStep(336, undefined, undefined),
new LearningStep(337, undefined, undefined),
new LearningStep(338, undefined, undefined),
new LearningStep(339, undefined, undefined),
new LearningStep(340, undefined, undefined),
new LearningStep(341, undefined, undefined),
```

```
new LearningStep(342, undefined, undefined),
new LearningStep(343, undefined, undefined),
new LearningStep(344, undefined, undefined),
new LearningStep(345, undefined, undefined),
new LearningStep(346, undefined, undefined),
new LearningStep(347, undefined, undefined),
new LearningStep(348, undefined, undefined),
new LearningStep(349, undefined, undefined),
new LearningStep(350, undefined, undefined),
new LearningStep(351, undefined, undefined),
new LearningStep(352, undefined, undefined),
new LearningStep(353, undefined, undefined),
new LearningStep(354, undefined, undefined),
new LearningStep(355, undefined, undefined),
new LearningStep(356, undefined, undefined),
new LearningStep(357, undefined, undefined),
new LearningStep(358, undefined, undefined),
new LearningStep(359, undefined, undefined),
new LearningStep(360, undefined, undefined),
new LearningStep(361, undefined, undefined),
new LearningStep(362, undefined, undefined),
new LearningStep(363, undefined, undefined),
new LearningStep(364, undefined, undefined),
new LearningStep(365, undefined, undefined),
new LearningStep(366, undefined, undefined),
new LearningStep(367, undefined, undefined),
new LearningStep(368, undefined, undefined),
new LearningStep(369, undefined, undefined),
new LearningStep(370, undefined, undefined),
new LearningStep(371, undefined, undefined),
new LearningStep(372, undefined, undefined),
new LearningStep(373, undefined, undefined),
new LearningStep(374, undefined, undefined),
new LearningStep(375, undefined, undefined),
new LearningStep(376, undefined, undefined),

new LearningStep(
  376,
  "Now we find the cheapest one.    We'll loop over the array
       , and\n   remember the lowest_price we have\n   seen
       so far. If we find a new ac\n   with a lower price,
       we'll store\n   the new lowest price, and store\n
       that AC as the best one.\n   \n   We'll start by
       setting the best\n   price very high. This is a trick
       \n   programmers use to make this\n   code simpler
       and shorter.",
  undefined ),

new LearningStep(377, undefined, undefined),
new LearningStep(378, undefined, undefined),
new LearningStep(379, undefined, undefined),
new LearningStep(380, undefined, undefined),
new LearningStep(381, undefined, undefined),
new LearningStep(382, undefined, undefined),
new LearningStep(383, undefined, undefined),
new LearningStep(384, undefined, undefined),
new LearningStep(385, undefined, undefined),
new LearningStep(386, undefined, undefined),
new LearningStep(387, undefined, undefined),
new LearningStep(388, undefined, undefined),
new LearningStep(389, undefined, undefined),
new LearningStep(390, undefined, undefined),
new LearningStep(391, undefined, undefined),
new LearningStep(392, undefined, undefined),
new LearningStep(393, undefined, undefined),
new LearningStep(394, undefined, undefined),
new LearningStep(395, undefined, undefined),
new LearningStep(396, undefined, undefined),
new LearningStep(397, undefined, undefined),
new LearningStep(398, undefined, undefined),
new LearningStep(399, undefined, undefined),
new LearningStep(400, undefined, undefined),
new LearningStep(401, undefined, undefined),
new LearningStep(402, undefined, undefined),
new LearningStep(403, undefined, undefined),
new LearningStep(404, undefined, undefined),
new LearningStep(405, undefined, undefined),
new LearningStep(406, undefined, undefined),
new LearningStep(407, undefined, undefined),
new LearningStep(408, undefined, undefined),
new LearningStep(409, undefined, undefined),
new LearningStep(410, undefined, undefined),
new LearningStep(411, undefined, undefined),
new LearningStep(412, undefined, undefined),
new LearningStep(413, undefined, undefined),
new LearningStep(414, undefined, undefined),
new LearningStep(415, undefined, undefined),
new LearningStep(416, undefined, undefined),
new LearningStep(417, undefined, undefined),
new LearningStep(418, undefined, undefined),
new LearningStep(419, undefined, undefined),
new LearningStep(420, undefined, undefined),
```

```
new LearningStep(421, undefined, undefined),
new LearningStep(422, undefined, undefined),
new LearningStep(423, undefined, undefined),
new LearningStep(424, undefined, undefined),
new LearningStep(425, undefined, undefined),
new LearningStep(426, undefined, undefined),
new LearningStep(427, undefined, undefined),
new LearningStep(428, undefined, undefined),
new LearningStep(429, undefined, undefined),
new LearningStep(430, undefined, undefined),
new LearningStep(431, undefined, undefined),
new LearningStep(432, undefined, undefined),
new LearningStep(433, undefined, undefined),
new LearningStep(434, undefined, undefined),
new LearningStep(435, undefined, undefined),
new LearningStep(436, undefined, undefined),
new LearningStep(437, undefined, undefined),
new LearningStep(438, undefined, undefined),
new LearningStep(439, undefined, undefined),
new LearningStep(440, undefined, undefined),
new LearningStep(441, undefined, undefined),
new LearningStep(442, undefined, undefined),
new LearningStep(443, undefined, undefined),
new LearningStep(444, undefined, undefined),
new LearningStep(445, undefined, undefined),
new LearningStep(446, undefined, undefined),
new LearningStep(447, undefined, undefined),
new LearningStep(448, undefined, undefined),
new LearningStep(449, undefined, undefined),
new LearningStep(450, undefined, undefined),
new LearningStep(451, undefined, undefined),
new LearningStep(452, undefined, undefined),
new LearningStep(453, undefined, undefined),
new LearningStep(454, undefined, undefined),
new LearningStep(455, undefined, undefined),
new LearningStep(456, undefined, undefined),
new LearningStep(457, undefined, undefined),
new LearningStep(458, undefined, undefined),
new LearningStep(459, undefined, undefined),
new LearningStep(460, undefined, undefined),
new LearningStep(461, undefined, undefined),
new LearningStep(462, undefined, undefined),
new LearningStep(463, undefined, undefined),
new LearningStep(464, undefined, undefined),
new LearningStep(465, undefined, undefined),
new LearningStep(466, undefined, undefined),
new LearningStep(467, undefined, undefined),
new LearningStep(468, undefined, undefined),
new LearningStep(469, undefined, undefined),
new LearningStep(470, undefined, undefined),
new LearningStep(471, undefined, undefined),
new LearningStep(472, undefined, undefined),
new LearningStep(473, undefined, undefined),
new LearningStep(474, undefined, undefined),
new LearningStep(475, undefined, undefined),
new LearningStep(476, undefined, undefined),
new LearningStep(477, undefined, undefined),
new LearningStep(478, undefined, undefined),
new LearningStep(479, undefined, undefined),
new LearningStep(480, undefined, undefined),
new LearningStep(481, undefined, undefined),

new LearningStep(
  482,
  " This is great!\nYou can imagine us using this\nprogram to
       recommend air conditioners\nfor thousands or
       millions of homes!\nAll it would take would be
       entering\nin an array with many homes in it,\nthen
       looping over that array and\nrecommending an AC for
       each house.\n\n
  We could also make a website where\npeople could enter data
       about\nthe rooms in their home, then we\ncould show
       recommended Air Conditioners\nor even sell them to
       them.\n\n
   If you could just make it easy\nfor people to find and
       access\nthis program, you could save\ntime for
       millions of people,\nlet them spend the money they\
       nsaved somewhere else,\nand help them live more\
       ncomfortably.\n\n
   Onwards!",
   undefined ),

new LearningStep(483, undefined, undefined)]
                  ) /* end LearningSteps */
                  ) /* end Action */]),
new KnowledgeUnit("Functions - how to store and reuse code",
          "
Functions allow us to store a list
of instructions and execute it.

The only thing about functions that is
new is how variables are handled inside
of functions. Otherwise they work
like an expression - they leave a value
on the stack.

In order to use the code inside a function,
we "call" the function. This saves
the prior stack and namespace and creates a
new one to use just for that function call.
        ",
           [
new Action (
                  new Program('   function willBe5(){
    return 5;
};

willBe5();

willBe5() == 5;
```

```
5 == willBe5();

willBe5() + willBe5() == 10;



            `
            ) /* end Program */,
                    new LearningSteps(
        /* Learning Steps */
        [
  new LearningStep(
    0,
    "## Functions\n\n
    Programmers want to avoid duplicating code over and over
         again. Otherwise, if it needs to be **changed**, you
         have to change it in many places. You might\nforget
         to change it in one place, and **create an error**.
         How can we avoid duplicating code?\n\n
    ",
    undefined ),

  new LearningStep(
    0,
    "Remember **variables** and how they let us store a value?
         \n\n
    **Functions** allow us to store a list of instructions and
         execute it.\n",
    undefined ),

  new LearningStep(
    0,
    "The only thing about functions that is new is how
         variables are handled **inside** of functions. We'll
         get to that in a moment. First, let's see a function
         in action.\n \n\n
    ",
    undefined ),

  new LearningStep(
    0,
    "When we **declare** a function, the computer makes a place
          in the namespace for it. This is like creating a
         variable. This is how you declare a function.\n\n
    ",
    undefined ),

  new LearningStep(
    0,
    "",
    undefined ),

  new LearningStep(
    1,
    " ",
    undefined ),

  new LearningStep(
    1,
    " When we call a function, we can see the instructions
         inside it. That's why stepping over it here doesn't
         show anything inside it.\n\n
    We will call the function soon.",
    undefined ),

  new LearningStep(
    1,
    "The { } 's let the parser know where the code for the
         function begins and ends.\n\n
    The syntax for a function is:\n```\nfunction function_name
         (){\n   code goes here\n}\n```",
    undefined ),

  new LearningStep(
    1,
    "The syntax that language designers made\nup for **calling
         ** a function is\n**writing its name followed by ()
         **.\n\n
    There's a special statement, called a **return** statement.
          Watch how it works. It leaves a value on the stack.\
         n",
    undefined ),

  new LearningStep(
    1,
    "Watch closely as you step through calling the function.",
    undefined ),

  new LearningStep(1, undefined, undefined),

  new LearningStep(2, undefined, undefined),

  new LearningStep(3, undefined, undefined),

  new LearningStep(4, undefined, undefined),

  new LearningStep(5, undefined, undefined),

  new LearningStep(
    5,
    "Now we are executing the code inside the function.",
    undefined ),

  new LearningStep(
    5,
    " ",
    undefined ),

  new LearningStep(6, undefined, undefined),

  new LearningStep(
    7,
    " ",
    undefined ),

  new LearningStep(
    8,
    "",
    undefined ),

  new LearningStep(
    8,
    "Return removes the function's frame and puts the return
         value onto the stack. See how the return value is on
         the stack now.",
    undefined ),

  new LearningStep(
    8,
    " ",

    new Question(
      "What is on the stack?",
      [4,2,8],
      [] ) ),

  new LearningStep(9, undefined, undefined),

  new LearningStep(10, undefined, undefined),

  new LearningStep(11, undefined, undefined),

  new LearningStep(12, undefined, undefined),

  new LearningStep(13, undefined, undefined),

  new LearningStep(14, undefined, undefined),

  new LearningStep(15, undefined, undefined),

  new LearningStep(16, undefined, undefined),

  new LearningStep(17, undefined, undefined),

  new LearningStep(
    17,
    "",

    new Question(
      "What value is on the stack?",
      [false,-10,10],
      ["","The answer is a boolean, not a number.","The answer
           is a boolean, not a number."] ) ),

  new LearningStep(18, undefined, undefined),

  new LearningStep(19, undefined, undefined),

  new LearningStep(20, undefined, undefined),
```

```
new LearningStep(21, undefined, undefined),

new LearningStep(22, undefined, undefined),

new LearningStep(23, undefined, undefined),

new LearningStep(24, undefined, undefined),

new LearningStep(25, undefined, undefined),

new LearningStep(26, undefined, undefined),

new LearningStep(
  26,
  undefined,

  new Question(
    "What is on the stack?",
    [0,1,false],
    [] ) ),

new LearningStep(
  27,
  "Functions work a lot\nlike an expression - they leave a
       value\non the stack.",
  undefined ),

new LearningStep(27, undefined, undefined),

new LearningStep(28, undefined, undefined),

new LearningStep(29, undefined, undefined),

new LearningStep(30, undefined, undefined),

new LearningStep(31, undefined, undefined),

new LearningStep(32, undefined, undefined),

new LearningStep(33, undefined, undefined),

new LearningStep(
  33,
  "",

  new Question(
    "What value did the return statement leave on the stack?"
        ,
    [100,10,1],
    ["","",""] ) ),

new LearningStep(34, undefined, undefined),

new LearningStep(35, undefined, undefined),

new LearningStep(36, undefined, undefined),

new LearningStep(37, undefined, undefined),

new LearningStep(38, undefined, undefined),

new LearningStep(39, undefined, undefined),

new LearningStep(
  39,
  "",

  new Question(
    "What value did the second call to willBeFive leave on
         the stack?",
    [100,0,1],
    ["","",""] ) ),

new LearningStep(40, undefined, undefined),

new LearningStep(41, undefined, undefined),

new LearningStep(42, undefined, undefined),

new LearningStep(
  42,
  "Great! Onwards!",
  undefined )]
                ) /* end LearningSteps */
                ) /* end Action */,
```

```
new Action (
                new Program(`   function willBe5(){
    return 5;
};

function f1(first){
    return first;
};

f1(10);
f1(1);
f1(5);

var z = willBe5();
f1( z );

f1(willBe5());



          `
          ) /* end Program */,
                new LearningSteps(
       /* Learning Steps */
       [
  new LearningStep(
    0,
    "## Arguments\n\n
    Functions that always do the same thing aren't very useful
        .\n\n
    You can pass values to the function by **putting them on
        the stack**. To put them on the stack, you put them
        **inside** the parentheses.\n\n
    \n",
    undefined ),

  new LearningStep(
    1,
    " ",
    undefined ),

  new LearningStep(2, undefined, undefined),

  new LearningStep(3, undefined, undefined),

  new LearningStep(4, undefined, undefined),

  new LearningStep(5, undefined, undefined),

  new LearningStep(
    6,
    "Now it pushes 10 onto the stack.\n\n
    \n",
    undefined ),

  new LearningStep(7, undefined, undefined),

  new LearningStep(
    8,
    "The computer created a new **frame** for this function
        call. A frame is a combination of a **stack**, **
        namespace**, and **next instruction to execute**. If
        you look down to see first frame() thats the
         frame we had originally. \n\n
    \n",
    undefined ),

  new LearningStep(
    8,
    "This instruction will create a variable inside this **new
        ** namespace using the value on the stack.\n\n
    ",
    undefined ),

  new LearningStep(
    9,
    "Look in the current frame's namespace and see that first
        has value 10 now.\n\n
    It is 10 because 10 was passed into the function via the
        stack. It was written inside the parentheses in f1
        (10)",
    undefined ),

  new LearningStep(9, undefined, undefined),
```

```
new LearningStep(10, undefined, undefined),

new LearningStep(
  10,
  "This will look up the variable first in the current frame.
       What is the value of first here?",
  undefined ),

new LearningStep(11, undefined, undefined),

new LearningStep(
  12,
  "Now we are back. The call to f1(10) left the value 10 on
       the stack. Look at the stack and notice it.\n\n
  Notice that there is **NO** variable named first in the
       current namespace. ",
  undefined ),

new LearningStep(
  12,
  "The variable first was created in the function call's
       namespace. When the function call ends that namespace
        is **discarded**.",
  undefined ),

new LearningStep(13, undefined, undefined),

new LearningStep(14, undefined, undefined),

new LearningStep(15, undefined, undefined),

new LearningStep(16, undefined, undefined),

new LearningStep(
  17,
  "Because 1 is passed in on the stack, first will get the
       value 1.",
  undefined ),

new LearningStep(18, undefined, undefined),

new LearningStep(
  18,
  "Notice that first **DOES NOT** have the value 10 from
       before. This is a new call of function f1 - each call
        to a function is **independent and separate**.",
  undefined ),

new LearningStep(19, undefined, undefined),

new LearningStep(20, undefined, undefined),

new LearningStep(21, undefined, undefined),

new LearningStep(22, undefined, undefined),

new LearningStep(23, undefined, undefined),

new LearningStep(24, undefined, undefined),

new LearningStep(25, undefined, undefined),

new LearningStep(26, undefined, undefined),

new LearningStep(27, undefined, undefined),

new LearningStep(28, undefined, undefined),

new LearningStep(29, undefined, undefined),

new LearningStep(30, undefined, undefined),

new LearningStep(
  30,
  "",

  new Question(
    "What value did f1(5) leave on the stack?",
    [100,0,10],
    ["","",""] ) ),

new LearningStep(
  31,
  "Functions can be used anywhere you can put an expression.
       They work effectively the same as an expression -
       they leave a value on the stack.",
  undefined ),

new LearningStep(31, undefined, undefined),

new LearningStep(32, undefined, undefined),

new LearningStep(33, undefined, undefined),

new LearningStep(34, undefined, undefined),

new LearningStep(35, undefined, undefined),

new LearningStep(36, undefined, undefined),

new LearningStep(37, undefined, undefined),

new LearningStep(38, undefined, undefined),

new LearningStep(39, undefined, undefined),

new LearningStep(40, undefined, undefined),

new LearningStep(41, undefined, undefined),

new LearningStep(42, undefined, undefined),

new LearningStep(43, undefined, undefined),

new LearningStep(44, undefined, undefined),

new LearningStep(45, undefined, undefined),

new LearningStep(46, undefined, undefined),

new LearningStep(47, undefined, undefined),

new LearningStep(48, undefined, undefined),

new LearningStep(49, undefined, undefined),

new LearningStep(50, undefined, undefined),

new LearningStep(51, undefined, undefined),

new LearningStep(52, undefined, undefined),

new LearningStep(53, undefined, undefined),

new LearningStep(54, undefined, undefined),

new LearningStep(55, undefined, undefined),

new LearningStep(56, undefined, undefined),

new LearningStep(57, undefined, undefined),

new LearningStep(58, undefined, undefined),

new LearningStep(59, undefined, undefined),

new LearningStep(60, undefined, undefined),

new LearningStep(61, undefined, undefined),

new LearningStep(62, undefined, undefined),

new LearningStep(63, undefined, undefined),

new LearningStep(
  63,
  undefined,

  new Question(
    "What is on the stack?",
    [1,4,3],
    [] ) ),

new LearningStep(
  63,
  "Great! willBe5() left the value 5 on the stack, which then
       got passed to the function f1. There it was assigned
       to the variable first.",
  undefined )]
                ) /* end LearningSteps */
```

```
                    ) /* end Action */,
new Action (
                    new Program(`    var outside = 100;

function another_f(){
    var outside = 10;
    return outside;
};

outside;

another_f();

outside;

outside = 50;

another_f() == 10;



            `
            ) /* end Program */,
                    new LearningSteps(
        /* Learning Steps */
        [
  new LearningStep(
    0,
    "## Function variables\n\n
    What if you have a function that\nmakes a variable with the
          **same name** as a variable outside the function?\n\
        n
    ",
    undefined ),

  new LearningStep(
    0,
    "We want\nchanges to variables inside the \nfunction to **
        not change variables** outside the function with the
        same name. We **also don't** want them to change the
        values of variables in **other functions**.\n\n
    \nOtherwise it would be a **mess**\nfiguring out what
        happens in a \nprogram, especially a large one that
        hundreds of people work on. \n\n
    You'd need\na list of all the names already used\nin order
        to not avoid changing\nto part of someone else's
        program as it runs. ",
    undefined ),

  new LearningStep(
    0,
    "The language does this by making a  new stack and
        namespace when you call a function. The combination
        of a stack and a namespace and a next instruction is
        called a **frame**.",
    undefined ),

  new LearningStep(0, undefined, undefined),

  new LearningStep(1, undefined, undefined),

  new LearningStep(2, undefined, undefined),

  new LearningStep(3, undefined, undefined),

  new LearningStep(4, undefined, undefined),

  new LearningStep(5, undefined, undefined),

  new LearningStep(6, undefined, undefined),

  new LearningStep(7, undefined, undefined),

  new LearningStep(8, undefined, undefined),

  new LearningStep(9, undefined, undefined),

  new LearningStep(10, undefined, undefined),

  new LearningStep(11, undefined, undefined),

  new LearningStep(
    12,
    "Notice the variable outside does **NOT** exist inside the
        function another_f yet. ",
    undefined ),

  new LearningStep(13, undefined, undefined),

  new LearningStep(14, undefined, undefined),

  new LearningStep(15, undefined, undefined),

  new LearningStep(16, undefined, undefined),

  new LearningStep(17, undefined, undefined),

  new LearningStep(18, undefined, undefined),

  new LearningStep(19, undefined, undefined),

  new LearningStep(
    20,
    "Look at the variable outside in the first frame. It **
        still** has the **same** value it had when we called
        another_f.",
    undefined ),

  new LearningStep(21, undefined, undefined),

  new LearningStep(
    21,
    "",

    new Question(
      "What value does outside have in this frame?",
      [10,0,5],
      ["","",""] ) ),

  new LearningStep(
    22,
    "Notice that outside here has the same value still - look
        at it in the current namespace.",
    undefined ),

  new LearningStep(23, undefined, undefined),

  new LearningStep(24, undefined, undefined),

  new LearningStep(25, undefined, undefined),

  new LearningStep(26, undefined, undefined),

  new LearningStep(27, undefined, undefined),

  new LearningStep(28, undefined, undefined),

  new LearningStep(29, undefined, undefined),

  new LearningStep(30, undefined, undefined),

  new LearningStep(31, undefined, undefined),

  new LearningStep(32, undefined, undefined),

  new LearningStep(33, undefined, undefined),

  new LearningStep(34, undefined, undefined),

  new LearningStep(35, undefined, undefined),

  new LearningStep(
    35,
    "",

    new Question(
      "What value does outside have in this frame?",
      [5,1,100],
      ["","",""] ) ),

  new LearningStep(36, undefined, undefined),

  new LearningStep(37, undefined, undefined),

  new LearningStep(38, undefined, undefined),

  new LearningStep(39, undefined, undefined),

  new LearningStep(
    39,
```

```
    undefined,

    new Question(
      "What is on the stack?",
      [0,1,false],
      [] ) ),

  new LearningStep(
    39,
    "Great! Function calls are designed to be **independent**
         by default.",
    undefined )]
                    ) /* end LearningSteps */
                    ) /* end Action */,
new Action (
                    new Program(`   function add_one(my_x){
  return my_x + 1;
};

add_one(10);

add_one(11);

add_one(add_one(10)) == 12;

var y = 10;

function f2(my_x){
  var y = 5;
  return my_x + 1;
};

y;

f2(100);

y;



            `
            ) /* end Program */,
                    new LearningSteps(
        /* Learning Steps */
        [
  new LearningStep(
    0,
    "## Arguments Review\n\n
    This may sound a bit redundant but this is such an
         important point.\n\n
    Because functions have a separate namespace, changing
         arguments inside the function doesn't usually affect
         what is outside the function. \n\n
    People try to write\nfunctions to be **independent**.\nThat
          way, they can do something reliably, that only
         depends on the values passed to it (also called **
         arguments**).",
    undefined ),

  new LearningStep(
    0,
    "There are ways to **force** a function to look outside of
         its own namespace but **none** of functions we cover
         will do that. The way to do it is to use the name of
         a variable that isn't an argument and isn't defined
         inside the function. Then the function will look
         outside via the **this** variable, which is managed
         by the language to refer to the namespace the
         function was called from.",
    undefined ),

  new LearningStep(0, undefined, undefined),

  new LearningStep(1, undefined, undefined),

  new LearningStep(2, undefined, undefined),

  new LearningStep(3, undefined, undefined),

  new LearningStep(4, undefined, undefined),

  new LearningStep(5, undefined, undefined),

  new LearningStep(6, undefined, undefined),

  new LearningStep(
    7,
    "Look in the namespace - my_x has the value passed to it (
         which is 10). \n\n
    It came from the 10 on the line add_one(10).",
    undefined ),

  new LearningStep(8, undefined, undefined),

  new LearningStep(9, undefined, undefined),

  new LearningStep(10, undefined, undefined),

  new LearningStep(11, undefined, undefined),

  new LearningStep(12, undefined, undefined),

  new LearningStep(13, undefined, undefined),

  new LearningStep(14, undefined, undefined),

  new LearningStep(15, undefined, undefined),

  new LearningStep(16, undefined, undefined),

  new LearningStep(17, undefined, undefined),

  new LearningStep(18, undefined, undefined),

  new LearningStep(19, undefined, undefined),

  new LearningStep(
    19,
    "",

    new Question(
      "What value is my_x?",
      [0,1,100],
      ["","","",""] ) ),

  new LearningStep(
    19,
    "It came from the 11 in add_one(11). It was passed to this
         function as an **argument** via the **stack**.",
    undefined ),

  new LearningStep(20, undefined, undefined),

  new LearningStep(21, undefined, undefined),

  new LearningStep(22, undefined, undefined),

  new LearningStep(
    23,
    "Functions can be used anywhere you can put an expression.
         When the computer sees a function call, it gets **
         evaluated** (reduced to a value on the stack). They
         work effectively the same as an expression - they
         leave a value on the stack.\n\n
    ",
    undefined ),

  new LearningStep(24, undefined, undefined),

  new LearningStep(25, undefined, undefined),

  new LearningStep(26, undefined, undefined),

  new LearningStep(27, undefined, undefined),

  new LearningStep(28, undefined, undefined),

  new LearningStep(29, undefined, undefined),

  new LearningStep(30, undefined, undefined),

  new LearningStep(31, undefined, undefined),

  new LearningStep(32, undefined, undefined),

  new LearningStep(33, undefined, undefined),

  new LearningStep(34, undefined, undefined),

  new LearningStep(35, undefined, undefined),
```

```
new LearningStep(36, undefined, undefined),

new LearningStep(
  36,
  "",

  new Question(
    "What value did add_one(10) leave on the stack?",
    [0,1,100],
    ["","",""] ) ),

new LearningStep(37, undefined, undefined),

new LearningStep(38, undefined, undefined),

new LearningStep(39, undefined, undefined),

new LearningStep(40, undefined, undefined),

new LearningStep(41, undefined, undefined),

new LearningStep(42, undefined, undefined),

new LearningStep(43, undefined, undefined),

new LearningStep(44, undefined, undefined),

new LearningStep(
  45,
  "Functions can be used anywhere you can put an expression.\
       n\n
  In this past line, you effectively did \n```\nadd_one( 11 )
       == 12;\n```\n\n
  since add_one(10) left 11 on the stack.",
  undefined ),

new LearningStep(46, undefined, undefined),

new LearningStep(47, undefined, undefined),

new LearningStep(48, undefined, undefined),

new LearningStep(49, undefined, undefined),

new LearningStep(50, undefined, undefined),

new LearningStep(51, undefined, undefined),

new LearningStep(52, undefined, undefined),

new LearningStep(53, undefined, undefined),

new LearningStep(54, undefined, undefined),

new LearningStep(
  54,
  undefined,

  new Question(
    "What is on the stack?",
    [0,11,5],
    [] ) ),

new LearningStep(55, undefined, undefined),

new LearningStep(56, undefined, undefined),

new LearningStep(57, undefined, undefined),

new LearningStep(58, undefined, undefined),

new LearningStep(59, undefined, undefined),

new LearningStep(60, undefined, undefined),

new LearningStep(61, undefined, undefined),

new LearningStep(62, undefined, undefined),

new LearningStep(63, undefined, undefined),

new LearningStep(64, undefined, undefined),

new LearningStep(65, undefined, undefined),

new LearningStep(66, undefined, undefined),

new LearningStep(67, undefined, undefined),

new LearningStep(68, undefined, undefined),

new LearningStep(69, undefined, undefined),

new LearningStep(70, undefined, undefined),

new LearningStep(
  71,
  "Look how the function\ndoesn't change the variable\
       noutside of it; that's because\nits namespace is
       separate\nand the function declared y inside itself."
       ,
  undefined ),

new LearningStep(71, undefined, undefined),

new LearningStep(
  72,
  "Great! Onwards!",
  undefined )]
                  ) /* end LearningSteps */
                  ) /* end Action */,
new Action (
                  new Program(`   var my_x = 1000;
var my_y = 10000;

function f3(my_x){
  my_x = 50;
  return my_x;
};

var result = f3(20);
20;
result;
my_x;

var result2 = f3(my_x);

my_x;



function adds(my_x, x2){
  return my_x + x2;
};

adds(5,2);
adds(5,2) == 5 + 2;



          `
          ) /* end Program */,
                  new LearningSteps(
      /* Learning Steps */
      [
  new LearningStep(
    0,
    "## Multiple Arguments \n\n
    By making functions independent by default, you can write
       functions\nand share them with your friends\nand you
       have the chance to\nmake them all work together
       without \nhaving to change them.",
    undefined ),

  new LearningStep(0, undefined, undefined),

  new LearningStep(1, undefined, undefined),

  new LearningStep(2, undefined, undefined),

  new LearningStep(3, undefined, undefined),

  new LearningStep(4, undefined, undefined),

  new LearningStep(5, undefined, undefined),

  new LearningStep(6, undefined, undefined),

  new LearningStep(7, undefined, undefined),
```

```
new LearningStep(8, undefined, undefined),

new LearningStep(9, undefined, undefined),

new LearningStep(10, undefined, undefined),

new LearningStep(11, undefined, undefined),

new LearningStep(12, undefined, undefined),

new LearningStep(13, undefined, undefined),

new LearningStep(14, undefined, undefined),

new LearningStep(15, undefined, undefined),

new LearningStep(16, undefined, undefined),

new LearningStep(17, undefined, undefined),

new LearningStep(18, undefined, undefined),

new LearningStep(19, undefined, undefined),

new LearningStep(20, undefined, undefined),

new LearningStep(21, undefined, undefined),

new LearningStep(22, undefined, undefined),

new LearningStep(23, undefined, undefined),

new LearningStep(
  23,
  "",

  new Question(
    "What value does my_x have in this frame?",
    [100,1,0],
    ["","",""] ) ),

new LearningStep(24, undefined, undefined),

new LearningStep(25, undefined, undefined),

new LearningStep(26, undefined, undefined),

new LearningStep(27, undefined, undefined),

new LearningStep(28, undefined, undefined),

new LearningStep(29, undefined, undefined),

new LearningStep(30, undefined, undefined),

new LearningStep(31, undefined, undefined),

new LearningStep(32, undefined, undefined),

new LearningStep(
  32,
  "",

  new Question(
    "What value does my_x have in this frame?",
    [0,100,1],
    ["","",""] ) ),

new LearningStep(33, undefined, undefined),

new LearningStep(
  33,
  "Try mentally executing this line before stepping through
       it.",
  undefined ),

new LearningStep(34, undefined, undefined),

new LearningStep(35, undefined, undefined),

new LearningStep(36, undefined, undefined),

new LearningStep(37, undefined, undefined),

new LearningStep(38, undefined, undefined),

new LearningStep(39, undefined, undefined),

new LearningStep(40, undefined, undefined),

new LearningStep(41, undefined, undefined),

new LearningStep(42, undefined, undefined),

new LearningStep(
  42,
  "",

  new Question(
    "What value does my_x have in this frame now that this
         line has executed?",
    [100,5,10],
    ["","",""] ) ),

new LearningStep(43, undefined, undefined),

new LearningStep(44, undefined, undefined),

new LearningStep(45, undefined, undefined),

new LearningStep(46, undefined, undefined),

new LearningStep(47, undefined, undefined),

new LearningStep(48, undefined, undefined),

new LearningStep(49, undefined, undefined),

new LearningStep(
  49,
  "my_x still has not changed in the first frame",
  undefined ),

new LearningStep(
  50,
  "still is 1000",
  undefined ),

new LearningStep(
  50,
  "## Two argument function\nIt works the same way, it just
       takes more values from the stack to put into it's
       argument variables. Here's an example. ",
  undefined ),

new LearningStep(50, undefined, undefined),

new LearningStep(51, undefined, undefined),

new LearningStep(52, undefined, undefined),

new LearningStep(53, undefined, undefined),

new LearningStep(54, undefined, undefined),

new LearningStep(55, undefined, undefined),

new LearningStep(56, undefined, undefined),

new LearningStep(57, undefined, undefined),

new LearningStep(58, undefined, undefined),

new LearningStep(
  58,
  "The first instruction of the function took 5 off the stack
       and stored it in my_x. This instruction will take 2
       off the stack and store it in x2.\n\n
  The first steps of a function are always to assign the
       values on the stack to its argument variables (if it
       has any).",
  undefined ),

new LearningStep(59, undefined, undefined),

new LearningStep(60, undefined, undefined),

new LearningStep(61, undefined, undefined),
```

```
  new LearningStep(62, undefined, undefined),

  new LearningStep(63, undefined, undefined),

  new LearningStep(64, undefined, undefined),

  new LearningStep(
    64,
    "",

    new Question(
      "What value is on the stack from adds(5,2)?",
      [5,10,0],
      ["","",""] ) ),

  new LearningStep(65, undefined, undefined),

  new LearningStep(
    65,
    "Try mentally executing this function call before stepping
         through it.",
    undefined ),

  new LearningStep(66, undefined, undefined),

  new LearningStep(67, undefined, undefined),

  new LearningStep(68, undefined, undefined),

  new LearningStep(69, undefined, undefined),

  new LearningStep(70, undefined, undefined),

  new LearningStep(71, undefined, undefined),

  new LearningStep(72, undefined, undefined),

  new LearningStep(73, undefined, undefined),

  new LearningStep(74, undefined, undefined),

  new LearningStep(75, undefined, undefined),

  new LearningStep(76, undefined, undefined),

  new LearningStep(77, undefined, undefined),

  new LearningStep(78, undefined, undefined),

  new LearningStep(79, undefined, undefined),

  new LearningStep(80, undefined, undefined),

  new LearningStep(
    81,
    "Great job! Onwards!",
    undefined )]
                    ) /* end LearningSteps */
                    ) /* end Action */,
new Action (
                    new Program(`


function subtract_one(my_x){
  return my_x - 1;
};

subtract_one(10);

subtract_one(3);

function times_two(my_x){
  return my_x * 2;
};

times_two(1);
var y = 2;
y = times_two(y) + subtract_one(y);
var z1 = times_two( times_two(y) );
var z2 = times_two ( subtract_one ( times_two( y ) ) );



          `
          ) /* end Program */,
                    new LearningSteps(
      /* Learning Steps */
      [
new LearningStep(
  0,
  "## Functions\nLet's take a look at some other functions
       for practice!\n\n
  This function subtracts 1 from it's argument. It's good to
       name functions descriptively; other people can **
       understand** your programs more easily this way. It
       also makes it easier for **you** to understand when
       you read it later!",
  undefined ),

new LearningStep(0, undefined, undefined),

new LearningStep(1, undefined, undefined),

new LearningStep(2, undefined, undefined),

new LearningStep(3, undefined, undefined),

new LearningStep(4, undefined, undefined),

new LearningStep(5, undefined, undefined),

new LearningStep(6, undefined, undefined),

new LearningStep(7, undefined, undefined),

new LearningStep(8, undefined, undefined),

new LearningStep(9, undefined, undefined),

new LearningStep(10, undefined, undefined),

new LearningStep(11, undefined, undefined),

new LearningStep(12, undefined, undefined),

new LearningStep(
  12,
  undefined,

  new Question(
    "What is on the stack?",
    [4,2,8],
    [] ) ),

new LearningStep(13, undefined, undefined),

new LearningStep(14, undefined, undefined),

new LearningStep(15, undefined, undefined),

new LearningStep(16, undefined, undefined),

new LearningStep(17, undefined, undefined),

new LearningStep(18, undefined, undefined),

new LearningStep(19, undefined, undefined),

new LearningStep(
  19,
  "",

  new Question(
    "What value does my_x have?",
    [5,0,10],
    ["","",""] ) ),

new LearningStep(20, undefined, undefined),

new LearningStep(21, undefined, undefined),

new LearningStep(22, undefined, undefined),

new LearningStep(23, undefined, undefined),

new LearningStep(
  23,
  undefined,
```

```
  new Question(
    "What is on the stack?",
    [4,6,8],
    [] ) ),

new LearningStep(
  24,
  "This function multiplies by 2.",
  undefined ),

new LearningStep(24, undefined, undefined),

new LearningStep(25, undefined, undefined),

new LearningStep(26, undefined, undefined),

new LearningStep(27, undefined, undefined),

new LearningStep(28, undefined, undefined),

new LearningStep(29, undefined, undefined),

new LearningStep(30, undefined, undefined),

new LearningStep(31, undefined, undefined),

new LearningStep(32, undefined, undefined),

new LearningStep(33, undefined, undefined),

new LearningStep(34, undefined, undefined),

new LearningStep(35, undefined, undefined),

new LearningStep(36, undefined, undefined),

new LearningStep(
  36,
  undefined,

  new Question(
    "What is on the stack?",
    [4,1,10],
    [] ) ),

new LearningStep(37, undefined, undefined),

new LearningStep(38, undefined, undefined),

new LearningStep(39, undefined, undefined),

new LearningStep(40, undefined, undefined),

new LearningStep(41, undefined, undefined),

new LearningStep(42, undefined, undefined),

new LearningStep(43, undefined, undefined),

new LearningStep(44, undefined, undefined),

new LearningStep(45, undefined, undefined),

new LearningStep(46, undefined, undefined),

new LearningStep(47, undefined, undefined),

new LearningStep(48, undefined, undefined),

new LearningStep(49, undefined, undefined),

new LearningStep(50, undefined, undefined),

new LearningStep(51, undefined, undefined),

new LearningStep(52, undefined, undefined),

new LearningStep(53, undefined, undefined),

new LearningStep(54, undefined, undefined),

new LearningStep(
  55,
  "",

  new Question(
    "What value did times_two(y) leave on the stack?",
    [100,1,5],
    ["","",""] ) ),

new LearningStep(55, undefined, undefined),

new LearningStep(56, undefined, undefined),

new LearningStep(57, undefined, undefined),

new LearningStep(58, undefined, undefined),

new LearningStep(59, undefined, undefined),

new LearningStep(60, undefined, undefined),

new LearningStep(61, undefined, undefined),

new LearningStep(
  62,
  "",

  new Question(
    "What value did subtract_one(y) leave on the stack?",
    [100,10,0],
    ["","",""] ) ),

new LearningStep(62, undefined, undefined),

new LearningStep(
  63,
  "Now y gets assigned its value.",
  undefined ),

new LearningStep(64, undefined, undefined),

new LearningStep(
  65,
  "Each **call** of a function gets its own **frame**, so
       they are independent when they only access variables
       inside their namespace.\n\n
  You *can* write functions that reference variables outside
       the function, but that makes them less self-contained
       . None of these functions do that.",
  undefined ),

new LearningStep(66, undefined, undefined),

new LearningStep(67, undefined, undefined),

new LearningStep(68, undefined, undefined),

new LearningStep(69, undefined, undefined),

new LearningStep(70, undefined, undefined),

new LearningStep(71, undefined, undefined),

new LearningStep(72, undefined, undefined),

new LearningStep(73, undefined, undefined),

new LearningStep(74, undefined, undefined),

new LearningStep(75, undefined, undefined),

new LearningStep(76, undefined, undefined),

new LearningStep(77, undefined, undefined),

new LearningStep(78, undefined, undefined),

new LearningStep(
  79,
  "",

  new Question(
    "What value did times_two(y) leave on the stack?",
    [5,1,100],
    ["","",""] ) ),

new LearningStep(
  79,
  "Now we call times_two again, using the value that was on
```

```
        the stack.",
    undefined ),

new LearningStep(79, undefined, undefined),

new LearningStep(80, undefined, undefined),

new LearningStep(81, undefined, undefined),

new LearningStep(82, undefined, undefined),

new LearningStep(83, undefined, undefined),

new LearningStep(84, undefined, undefined),

new LearningStep(85, undefined, undefined),

new LearningStep(86, undefined, undefined),

new LearningStep(87, undefined, undefined),

new LearningStep(
  87,
  "",

  new Question(
    "What is the value of z1?",
    [0,5,100],
    ["","",""] ) ),

new LearningStep(88, undefined, undefined),

new LearningStep(89, undefined, undefined),

new LearningStep(90, undefined, undefined),

new LearningStep(91, undefined, undefined),

new LearningStep(92, undefined, undefined),

new LearningStep(93, undefined, undefined),

new LearningStep(94, undefined, undefined),

new LearningStep(95, undefined, undefined),

new LearningStep(96, undefined, undefined),

new LearningStep(97, undefined, undefined),

new LearningStep(98, undefined, undefined),

new LearningStep(99, undefined, undefined),

new LearningStep(100, undefined, undefined),

new LearningStep(101, undefined, undefined),

new LearningStep(102, undefined, undefined),

new LearningStep(103, undefined, undefined),

new LearningStep(
  104,
  "",

  new Question(
    "What value did times_two(y) leave on the stack?",
    [0,1,100],
    ["","",""] ) ),

new LearningStep(104, undefined, undefined),

new LearningStep(105, undefined, undefined),

new LearningStep(106, undefined, undefined),

new LearningStep(107, undefined, undefined),

new LearningStep(108, undefined, undefined),

new LearningStep(109, undefined, undefined),

new LearningStep(110, undefined, undefined),

new LearningStep(
  111,
  "",

  new Question(
    "What value did the call of subtract_one leave on the
        stack?",
    [10,1,100],
    ["","",""] ) ),

new LearningStep(111, undefined, undefined),

new LearningStep(112, undefined, undefined),

new LearningStep(113, undefined, undefined),

new LearningStep(114, undefined, undefined),

new LearningStep(115, undefined, undefined),

new LearningStep(116, undefined, undefined),

new LearningStep(117, undefined, undefined),

new LearningStep(118, undefined, undefined),

new LearningStep(
  119,
  "",

  new Question(
    "What is the value of z2?",
    [100,10,5],
    ["","",""] ) ),

new LearningStep(
  119,
  "Great job! Let's get some more practice with functions.",
  undefined )]
                    ) /* end LearningSteps */
                    ) /* end Action */,
new Action (
                    new Program(`  function divide_by_two(my_x)
                        {
  return my_x / 2;
};

var my_x = divide_by_two(100);

my_x = divide_by_two(3);


function add_to_ten(my_x) {
    while(my_x < 10) {
        my_x = my_x + 1;
    }
    return my_x;
}

add_to_ten(5);

function add_to_ten(my_x) {
    for(var i = my_x; i < 10; i++) {
        my_x = my_x + 1;
    }
    return my_x;
}

add_to_ten(7);


            `
            ) /* end Program */,
                    new LearningSteps(
        /* Learning Steps */
        [
  new LearningStep(
    0,
    "## Functions\nLet's review functions.",
    undefined ),

  new LearningStep(0, undefined, undefined),

  new LearningStep(1, undefined, undefined),
```

```
new LearningStep(2, undefined, undefined),

new LearningStep(3, undefined, undefined),

new LearningStep(4, undefined, undefined),

new LearningStep(5, undefined, undefined),

new LearningStep(6, undefined, undefined),

new LearningStep(7, undefined, undefined),

new LearningStep(8, undefined, undefined),

new LearningStep(9, undefined, undefined),

new LearningStep(10, undefined, undefined),

new LearningStep(11, undefined, undefined),

new LearningStep(
  12,
  "",
  undefined ),

new LearningStep(13, undefined, undefined),

new LearningStep(14, undefined, undefined),

new LearningStep(
  15,
  "",

  new Question(
    "What is the value of my_x?",
    [0,1,10],
    ["","",""] ) ),

new LearningStep(15, undefined, undefined),

new LearningStep(16, undefined, undefined),

new LearningStep(17, undefined, undefined),

new LearningStep(18, undefined, undefined),

new LearningStep(19, undefined, undefined),

new LearningStep(20, undefined, undefined),

new LearningStep(21, undefined, undefined),

new LearningStep(
  22,
  "",

  new Question(
    "What is the value of my_x?",
    [1,50,5],
    ["","That is the value my_x has in first frame, not the
         current frame. What is the current frame? What
         value was passed to this function?",""] ) ),

new LearningStep(22, undefined, undefined),

new LearningStep(23, undefined, undefined),

new LearningStep(
  24,
  "",
  undefined ),

new LearningStep(
  25,
  "",
  undefined ),

new LearningStep(
  26,
  "",
  undefined ),

new LearningStep(
  27,
  "",
  undefined ),

new LearningStep(
  28,
  "What will this function do? Sometimes the names can be
       deceptive. The computer **just executes the code in
       the function** and doesn't change its behavior based
       on the name.\n\n
  Try mentally executing the line\n```\nadd_to_ten(5);\n```\
       \n
  What will it leave on the stack?\n\n
  ",
  undefined ),

new LearningStep(
  28,
  "When you're done, step through the function and check your
       understanding. You can also hold down the right
       arrow key to step **all** the way through to the
       function leaving a value on the stack, then step **
       backwards** to see how the value got there.",
  undefined ),

new LearningStep(28, undefined, undefined),

new LearningStep(29, undefined, undefined),

new LearningStep(30, undefined, undefined),

new LearningStep(31, undefined, undefined),

new LearningStep(32, undefined, undefined),

new LearningStep(33, undefined, undefined),

new LearningStep(34, undefined, undefined),

new LearningStep(
  35,
  "",

  new Question(
    "What is about to go on the stack?",
    [1.5,10,1],
    ["That is the value of my_x in the first frame, not the
         current frame. What is the current frame? What
         value was passed to this function?","",""] ) ),

new LearningStep(35, undefined, undefined),

new LearningStep(36, undefined, undefined),

new LearningStep(37, undefined, undefined),

new LearningStep(38, undefined, undefined),

new LearningStep(39, undefined, undefined),

new LearningStep(40, undefined, undefined),

new LearningStep(41, undefined, undefined),

new LearningStep(42, undefined, undefined),

new LearningStep(43, undefined, undefined),

new LearningStep(44, undefined, undefined),

new LearningStep(45, undefined, undefined),

new LearningStep(46, undefined, undefined),

new LearningStep(47, undefined, undefined),

new LearningStep(48, undefined, undefined),

new LearningStep(49, undefined, undefined),

new LearningStep(50, undefined, undefined),

new LearningStep(51, undefined, undefined),

new LearningStep(52, undefined, undefined),

new LearningStep(53, undefined, undefined),
```

```
new LearningStep(54, undefined, undefined),

new LearningStep(55, undefined, undefined),

new LearningStep(56, undefined, undefined),

new LearningStep(57, undefined, undefined),

new LearningStep(58, undefined, undefined),

new LearningStep(59, undefined, undefined),

new LearningStep(60, undefined, undefined),

new LearningStep(61, undefined, undefined),

new LearningStep(62, undefined, undefined),

new LearningStep(63, undefined, undefined),

new LearningStep(64, undefined, undefined),

new LearningStep(65, undefined, undefined),

new LearningStep(66, undefined, undefined),

new LearningStep(67, undefined, undefined),

new LearningStep(68, undefined, undefined),

new LearningStep(69, undefined, undefined),

new LearningStep(70, undefined, undefined),

new LearningStep(71, undefined, undefined),

new LearningStep(72, undefined, undefined),

new LearningStep(73, undefined, undefined),

new LearningStep(74, undefined, undefined),

new LearningStep(75, undefined, undefined),

new LearningStep(76, undefined, undefined),

new LearningStep(77, undefined, undefined),

new LearningStep(78, undefined, undefined),

new LearningStep(79, undefined, undefined),

new LearningStep(80, undefined, undefined),

new LearningStep(81, undefined, undefined),

new LearningStep(82, undefined, undefined),

new LearningStep(83, undefined, undefined),

new LearningStep(84, undefined, undefined),

new LearningStep(85, undefined, undefined),

new LearningStep(86, undefined, undefined),

new LearningStep(87, undefined, undefined),

new LearningStep(88, undefined, undefined),

new LearningStep(89, undefined, undefined),

new LearningStep(90, undefined, undefined),

new LearningStep(91, undefined, undefined),

new LearningStep(
  92,
  "",

  new Question(
    "What value did the call to add_to_ten leave on the stack
        ?",
    [1,0,100],
    ["","","",""] ) ),

new LearningStep(92, undefined, undefined),

new LearningStep(
  93,
  "",

  new Question(
    "What is the value of my_x?",
    [0,10,1],
    ["","That was the value my_x had inside the previous
        function calls frame. We're back in the first
        frame now since the function call finished.",""] )
        ),

new LearningStep(
  93,
  "This function has a for loop in it.\nIt works the same as
      normal for loops\neven when it is inside a function."
      ,
  undefined ),

new LearningStep(
  93,
  "Oh, and by the way, here we're showing you a new unary
      operator **++**\n\n
  ```\ni++\n```\n\n
  works the same as\n\n
  ```\ni = i + 1\n```\n++ is called the **increment**
      operator, because it adds one to the value already in
       the variable it is next to.\n\n
  It's just a little shorter. These language designers are
      really lazy! Though when writing loops you do often
      write i = i + 1, so it helps. ",
  undefined),

new LearningStep(
  93,
  "Try mentally executing add_to_ten(7) and then step through
      .",
  undefined),

new LearningStep(94, undefined, undefined),

new LearningStep(95, undefined, undefined),

new LearningStep(96, undefined, undefined),

new LearningStep(97, undefined, undefined),

new LearningStep(98, undefined, undefined),

new LearningStep(99, undefined, undefined),

new LearningStep(100, undefined, undefined),

new LearningStep(101, undefined, undefined),

new LearningStep(102, undefined, undefined),

new LearningStep(103, undefined, undefined),

new LearningStep(104, undefined, undefined),

new LearningStep(105, undefined, undefined),

new LearningStep(106, undefined, undefined),

new LearningStep(107, undefined, undefined),

new LearningStep(108, undefined, undefined),

new LearningStep(109, undefined, undefined),

new LearningStep(110, undefined, undefined),

new LearningStep(111, undefined, undefined),

new LearningStep(112, undefined, undefined),

new LearningStep(113, undefined, undefined),

new LearningStep(114, undefined, undefined),
```

```
new LearningStep(115, undefined, undefined),

new LearningStep(116, undefined, undefined),

new LearningStep(117, undefined, undefined),

new LearningStep(
  118,
  "",

  new Question(
    "What did i++ change i's value to?",
    [0,1,100],
    ["","",""] ) ),

new LearningStep(118, undefined, undefined),

new LearningStep(119, undefined, undefined),

new LearningStep(120, undefined, undefined),

new LearningStep(121, undefined, undefined),

new LearningStep(122, undefined, undefined),

new LearningStep(123, undefined, undefined),

new LearningStep(124, undefined, undefined),

new LearningStep(125, undefined, undefined),

new LearningStep(126, undefined, undefined),

new LearningStep(127, undefined, undefined),

new LearningStep(128, undefined, undefined),

new LearningStep(129, undefined, undefined),

new LearningStep(130, undefined, undefined),

new LearningStep(131, undefined, undefined),

new LearningStep(132, undefined, undefined),

new LearningStep(133, undefined, undefined),

new LearningStep(134, undefined, undefined),

new LearningStep(135, undefined, undefined),

new LearningStep(136, undefined, undefined),

new LearningStep(137, undefined, undefined),

new LearningStep(138, undefined, undefined),

new LearningStep(139, undefined, undefined),

new LearningStep(140, undefined, undefined),

new LearningStep(141, undefined, undefined),

new LearningStep(142, undefined, undefined),

new LearningStep(143, undefined, undefined),

new LearningStep(144, undefined, undefined),

new LearningStep(145, undefined, undefined),

new LearningStep(146, undefined, undefined),

new LearningStep(147, undefined, undefined),

new LearningStep(148, undefined, undefined),

new LearningStep(149, undefined, undefined),

new LearningStep(150, undefined, undefined),

new LearningStep(151, undefined, undefined),

new LearningStep(152, undefined, undefined),

new LearningStep(153, undefined, undefined),

new LearningStep(154, undefined, undefined),

new LearningStep(155, undefined, undefined),

new LearningStep(156, undefined, undefined),

new LearningStep(157, undefined, undefined),

new LearningStep(158, undefined, undefined),

new LearningStep(
  159,
  "",
  undefined ),

new LearningStep(
  160,
  "",

  new Question(
    "What value did this function call leave on the stack?",
    [100,0,1],
    ["","",""] ) ),

new LearningStep(
  160,
  "Very good! Let's learn **recursion** next.",
  undefined )]
                    ) /* end LearningSteps */
                    ) /* end Action */,
new Action (
                    new Program(`
var a = [ 1, 10];
var total_of_a = 0;

for (var i = 0; i < a.length ; i = i + 1 ){
  total_of_a = a[i] + total_of_a;
}

function sum( i, the_array ) {
 if ( i < the_array.length){
    return a[i] + sum(i + 1, the_array);
 } else {
  return 0;
 }
}

var a = [];

sum(0, a);

var a = [1];
sum(0, a);

var a = [1, 10];
sum(0, a);

var a = [1, 10, 100];
sum(0, a);



            `
            ) /* end Program */,
                    new LearningSteps(
        /* Learning Steps */
        [
  new LearningStep(
    0,
    "## Recursion\n\n
    Let's first show a for loop that adds up numbers. Let's
         hurry to the exciting part that comes after!",
    undefined ),

  new LearningStep(1, undefined, undefined),

  new LearningStep(2, undefined, undefined),

  new LearningStep(3, undefined, undefined),

  new LearningStep(4, undefined, undefined),
```

```
new LearningStep(5, undefined, undefined),
new LearningStep(6, undefined, undefined),
new LearningStep(7, undefined, undefined),
new LearningStep(8, undefined, undefined),
new LearningStep(9, undefined, undefined),
new LearningStep(10, undefined, undefined),
new LearningStep(11, undefined, undefined),
new LearningStep(12, undefined, undefined),
new LearningStep(13, undefined, undefined),
new LearningStep(14, undefined, undefined),
new LearningStep(15, undefined, undefined),
new LearningStep(16, undefined, undefined),
new LearningStep(17, undefined, undefined),
new LearningStep(18, undefined, undefined),
new LearningStep(19, undefined, undefined),
new LearningStep(20, undefined, undefined),
new LearningStep(21, undefined, undefined),
new LearningStep(22, undefined, undefined),
new LearningStep(23, undefined, undefined),
new LearningStep(24, undefined, undefined),
new LearningStep(25, undefined, undefined),
new LearningStep(26, undefined, undefined),
new LearningStep(27, undefined, undefined),
new LearningStep(28, undefined, undefined),
new LearningStep(29, undefined, undefined),
new LearningStep(30, undefined, undefined),
new LearningStep(31, undefined, undefined),
new LearningStep(32, undefined, undefined),
new LearningStep(33, undefined, undefined),
new LearningStep(34, undefined, undefined),
new LearningStep(35, undefined, undefined),
new LearningStep(36, undefined, undefined),
new LearningStep(37, undefined, undefined),
new LearningStep(38, undefined, undefined),
new LearningStep(39, undefined, undefined),
new LearningStep(40, undefined, undefined),
new LearningStep(41, undefined, undefined),
new LearningStep(42, undefined, undefined),
new LearningStep(43, undefined, undefined),
new LearningStep(44, undefined, undefined),
new LearningStep(45, undefined, undefined),
new LearningStep(46, undefined, undefined),
new LearningStep(47, undefined, undefined),
new LearningStep(48, undefined, undefined),
new LearningStep(49, undefined, undefined),
new LearningStep(50, undefined, undefined),
new LearningStep(51, undefined, undefined),
new LearningStep(52, undefined, undefined),
new LearningStep(53, undefined, undefined),
new LearningStep(54, undefined, undefined),
new LearningStep(55, undefined, undefined),
new LearningStep(56, undefined, undefined),
new LearningStep(57, undefined, undefined),
new LearningStep(58, undefined, undefined),
new LearningStep(59, undefined, undefined),
new LearningStep(60, undefined, undefined),
new LearningStep(61, undefined, undefined),
new LearningStep(62, undefined, undefined),
new LearningStep(63, undefined, undefined),
new LearningStep(64, undefined, undefined),
new LearningStep(65, undefined, undefined),
new LearningStep(
  66,
  "We've already seen examples of functions that call **other
       ** functions. But a function can also call **itself
       **. We call this **recursion**.",
  undefined ),
new LearningStep(
  66,
  "We'll step through an example of a **recursive** function,
        then we'll talk more about the concept of recursion.
       ",
  undefined ),
new LearningStep(
  66,
  "The **important** takeaway is that recursive function
       calls work **the same as normal function calls**. The
        computer executes them the same way.\n\n
  It is just mind-bending for nearly everyone when you first
       see it. We're doing the most basic example here - a
       function that adds up all the numbers in an array.",
  undefined ),
new LearningStep(67, undefined, undefined),
new LearningStep(68, undefined, undefined),
new LearningStep(69, undefined, undefined),
new LearningStep(70, undefined, undefined),
new LearningStep(71, undefined, undefined),
new LearningStep(72, undefined, undefined),
new LearningStep(73, undefined, undefined),
new LearningStep(74, undefined, undefined),
new LearningStep(75, undefined, undefined),
new LearningStep(76, undefined, undefined),
new LearningStep(77, undefined, undefined),
new LearningStep(78, undefined, undefined),
```

```
new LearningStep(79, undefined, undefined),

new LearningStep(80, undefined, undefined),

new LearningStep(81, undefined, undefined),

new LearningStep(82, undefined, undefined),

new LearningStep(83, undefined, undefined),

new LearningStep(84, undefined, undefined),

new LearningStep(85, undefined, undefined),

new LearningStep(86, undefined, undefined),

new LearningStep(87, undefined, undefined),

new LearningStep(88, undefined, undefined),

new LearningStep(
  89,
  "The sum of an empty array is 0. Makes sense. Let's try an
       array of size 1.",
  undefined ),

new LearningStep(90, undefined, undefined),

new LearningStep(91, undefined, undefined),

new LearningStep(92, undefined, undefined),

new LearningStep(93, undefined, undefined),

new LearningStep(94, undefined, undefined),

new LearningStep(95, undefined, undefined),

new LearningStep(96, undefined, undefined),

new LearningStep(97, undefined, undefined),

new LearningStep(98, undefined, undefined),

new LearningStep(99, undefined, undefined),

new LearningStep(100, undefined, undefined),

new LearningStep(101, undefined, undefined),

new LearningStep(102, undefined, undefined),

new LearningStep(103, undefined, undefined),

new LearningStep(104, undefined, undefined),

new LearningStep(105, undefined, undefined),

new LearningStep(106, undefined, undefined),

new LearningStep(107, undefined, undefined),

new LearningStep(108, undefined, undefined),

new LearningStep(109, undefined, undefined),

new LearningStep(110, undefined, undefined),

new LearningStep(111, undefined, undefined),

new LearningStep(112, undefined, undefined),

new LearningStep(113, undefined, undefined),

new LearningStep(114, undefined, undefined),

new LearningStep(115, undefined, undefined),

new LearningStep(116, undefined, undefined),

new LearningStep(117, undefined, undefined),

new LearningStep(
  118,
  "This serves the same purpose as the i = i + 1 in the for
       loop. It effectively advances forward in the array.",
  undefined ),

new LearningStep(
  119,
  "",

  new Question(
    "What value did i + 1 put onto the stack?",
    [100,0,10],
    ["","",""] ) ),

new LearningStep(119, undefined, undefined),

new LearningStep(
  120,
  "Now we will call sum using a new frame. Count the number
       of frames there are right now.",
  undefined ),

new LearningStep(
  121,
  "Count the number of frames again. We've created a new
       frame for this call.",
  undefined ),

new LearningStep(
  121,
  "Notice that i in this frame will have the value 1 since it
        was passed on the stack.",
  undefined ),

new LearningStep(122, undefined, undefined),

new LearningStep(123, undefined, undefined),

new LearningStep(124, undefined, undefined),

new LearningStep(125, undefined, undefined),

new LearningStep(126, undefined, undefined),

new LearningStep(127, undefined, undefined),

new LearningStep(
  128,
  "",

  new Question(
    "Was i < the_array.length ?",
    [true,10,-10],
    ["","The answer is a boolean, not a number.","The answer
         is a boolean, not a number."] ) ),

new LearningStep(128, undefined, undefined),

new LearningStep(129, undefined, undefined),

new LearningStep(130, undefined, undefined),

new LearningStep(
  131,
  "Now this call to sum will return 0, since an array of
       length one has no element in position i (here i is 1
       and refers to the second position in the array).\n\n
  Look at all the frames again before we step and return from
       here.",
  undefined ),

new LearningStep(
  132,
  "Now the return value of the second call to sum is on the
       stack and there are only two frames.\n\n
  Each call to sum has a **separate namespace**, so they don'
       t directly affect each other.",
  undefined ),

new LearningStep(
  132,
  "",

  new Question(
    "What value did sum return that is now on the stack?",
    [1,10,5],
    ["What is the_array? What was i's value in the last call
```

```
        to sum?","What is the_array? What value was i in
        the last call?",""] ) ),

new LearningStep(
  132,
  "Now, this instruction will add 1 + 0 and return that value
      .",
  undefined ),

new LearningStep(133, undefined, undefined),

new LearningStep(134, undefined, undefined),

new LearningStep(
  134,
  "",

  new Question(
    "What value did sum(0,a) return and put on the stack?",
    [100,5,10],
    ["","What values did the_array have?","Step back to the
        final return step. the_array[0] got added to the
        sum of the rest of the array."] ) ),

new LearningStep(135, undefined, undefined),

new LearningStep(136, undefined, undefined),

new LearningStep(137, undefined, undefined),

new LearningStep(138, undefined, undefined),

new LearningStep(139, undefined, undefined),

new LearningStep(140, undefined, undefined),

new LearningStep(141, undefined, undefined),

new LearningStep(142, undefined, undefined),

new LearningStep(143, undefined, undefined),

new LearningStep(144, undefined, undefined),

new LearningStep(145, undefined, undefined),

new LearningStep(146, undefined, undefined),

new LearningStep(147, undefined, undefined),

new LearningStep(148, undefined, undefined),

new LearningStep(149, undefined, undefined),

new LearningStep(150, undefined, undefined),

new LearningStep(151, undefined, undefined),

new LearningStep(152, undefined, undefined),

new LearningStep(153, undefined, undefined),

new LearningStep(154, undefined, undefined),

new LearningStep(155, undefined, undefined),

new LearningStep(156, undefined, undefined),

new LearningStep(157, undefined, undefined),

new LearningStep(158, undefined, undefined),

new LearningStep(159, undefined, undefined),

new LearningStep(160, undefined, undefined),

new LearningStep(161, undefined, undefined),

new LearningStep(162, undefined, undefined),

new LearningStep(163, undefined, undefined),

new LearningStep(164, undefined, undefined),

new LearningStep(165, undefined, undefined),

new LearningStep(166, undefined, undefined),

new LearningStep(167, undefined, undefined),

new LearningStep(168, undefined, undefined),

new LearningStep(169, undefined, undefined),

new LearningStep(170, undefined, undefined),

new LearningStep(171, undefined, undefined),

new LearningStep(172, undefined, undefined),

new LearningStep(173, undefined, undefined),

new LearningStep(174, undefined, undefined),

new LearningStep(175, undefined, undefined),

new LearningStep(176, undefined, undefined),

new LearningStep(177, undefined, undefined),

new LearningStep(178, undefined, undefined),

new LearningStep(179, undefined, undefined),

new LearningStep(180, undefined, undefined),

new LearningStep(181, undefined, undefined),

new LearningStep(182, undefined, undefined),

new LearningStep(183, undefined, undefined),

new LearningStep(184, undefined, undefined),

new LearningStep(185, undefined, undefined),

new LearningStep(186, undefined, undefined),

new LearningStep(187, undefined, undefined),

new LearningStep(
  188,
  "Now we're in the third call to sum. Why is this if
      statement here?\n\n
  It acts as a check to decide when to stop calling sum. This
       kind of check to tell when to stop is called a **
      base case**. Without a **base case** a recursive
      function would keep calling itself forever.\n",
  undefined ),

new LearningStep(189, undefined, undefined),

new LearningStep(190, undefined, undefined),

new LearningStep(191, undefined, undefined),

new LearningStep(192, undefined, undefined),

new LearningStep(193, undefined, undefined),

new LearningStep(194, undefined, undefined),

new LearningStep(195, undefined, undefined),

new LearningStep(
  196,
  "Now, the function will return 0, go to the prior frame,
      add that to the value on the stack, and continue
      until it finishes and returns the sum.\n\n
  Step and watch this carefully.",
  undefined ),

new LearningStep(
  197,
  "Now we are back in the frame of the second call of sum.",
  undefined ),

new LearningStep(
  197,
  "See the value beneath zero on the stack? That is there
```

```
        from a[1] ( i is 1 in this frame)",
    undefined ),

new LearningStep(198, undefined, undefined),

new LearningStep(
    198,
    "Now we will return 10 via the stack to the prior frame.
        The prior frame is for the first call to sum.",
    undefined ),

new LearningStep(
    199,
    "We're back where we started, in the first call to sum.
        Look at the frames and check.",
    undefined ),

new LearningStep(
    199,
    "The 1 on the bottom of the stack is from a[0]. Try **
        stepping back** to the beginning of the call to sum (
        learning step **155**) then step through to the
        instruction that added the 1 on the stack you see
        here. (Hint: around learning step **166**)\n\n
    Then keep stepping forward to step **209** (this step).\n\n
    You can copy and paste this into a document if that helps."
        ,
    undefined ),

new LearningStep(200, undefined, undefined),

new LearningStep(
    201,
    "Now we're back to the original frame. Woah.",
    undefined ),

new LearningStep(
    201,
    "Recursion has a mathematical feel to it. You can think of
        it working like this for sum.\n\n
    ```\nsum of [1,10] = 1 + sum of [10]\nsum of [10] = 10 +
        sum of [ ]\nsum of [ ] = 0\n```\n\n
    ",
    undefined),

new LearningStep(202, undefined, undefined),

new LearningStep(203, undefined, undefined),

new LearningStep(204, undefined, undefined),

new LearningStep(205, undefined, undefined),

new LearningStep(206, undefined, undefined),

new LearningStep(207, undefined, undefined),

new LearningStep(208, undefined, undefined),

new LearningStep(209, undefined, undefined),

new LearningStep(210, undefined, undefined),

new LearningStep(210, undefined, undefined),

new LearningStep(211, undefined, undefined),

new LearningStep(212, undefined, undefined),

new LearningStep(213, undefined, undefined),

new LearningStep(214, undefined, undefined),

new LearningStep(215, undefined, undefined),

new LearningStep(216, undefined, undefined),

new LearningStep(217, undefined, undefined),

new LearningStep(218, undefined, undefined),

new LearningStep(219, undefined, undefined),

new LearningStep(220, undefined, undefined),

new LearningStep(221, undefined, undefined),

new LearningStep(222, undefined, undefined),

new LearningStep(223, undefined, undefined),

new LearningStep(224, undefined, undefined),

new LearningStep(225, undefined, undefined),

new LearningStep(226, undefined, undefined),

new LearningStep(227, undefined, undefined),

new LearningStep(228, undefined, undefined),

new LearningStep(229, undefined, undefined),

new LearningStep(230, undefined, undefined),

new LearningStep(231, undefined, undefined),

new LearningStep(232, undefined, undefined),

new LearningStep(233, undefined, undefined),

new LearningStep(234, undefined, undefined),

new LearningStep(235, undefined, undefined),

new LearningStep(236, undefined, undefined),

new LearningStep(237, undefined, undefined),

new LearningStep(238, undefined, undefined),

new LearningStep(239, undefined, undefined),

new LearningStep(240, undefined, undefined),

new LearningStep(241, undefined, undefined),

new LearningStep(242, undefined, undefined),

new LearningStep(243, undefined, undefined),

new LearningStep(244, undefined, undefined),

new LearningStep(245, undefined, undefined),

new LearningStep(246, undefined, undefined),

new LearningStep(247, undefined, undefined),

new LearningStep(248, undefined, undefined),

new LearningStep(249, undefined, undefined),

new LearningStep(250, undefined, undefined),

new LearningStep(251, undefined, undefined),

new LearningStep(252, undefined, undefined),

new LearningStep(253, undefined, undefined),

new LearningStep(254, undefined, undefined),

new LearningStep(255, undefined, undefined),

new LearningStep(256, undefined, undefined),

new LearningStep(257, undefined, undefined),

new LearningStep(258, undefined, undefined),

new LearningStep(259, undefined, undefined),

new LearningStep(260, undefined, undefined),

new LearningStep(261, undefined, undefined),

new LearningStep(262, undefined, undefined),

new LearningStep(263, undefined, undefined),
```
```

```
new LearningStep(264, undefined, undefined),

new LearningStep(265, undefined, undefined),

new LearningStep(266, undefined, undefined),

new LearningStep(267, undefined, undefined),

new LearningStep(268, undefined, undefined),

new LearningStep(269, undefined, undefined),

new LearningStep(270, undefined, undefined),

new LearningStep(271, undefined, undefined),

new LearningStep(272, undefined, undefined),

new LearningStep(273, undefined, undefined),

new LearningStep(274, undefined, undefined),

new LearningStep(275, undefined, undefined),

new LearningStep(276, undefined, undefined),

new LearningStep(277, undefined, undefined),

new LearningStep(278, undefined, undefined),

new LearningStep(279, undefined, undefined),

new LearningStep(280, undefined, undefined),

new LearningStep(281, undefined, undefined),

new LearningStep(282, undefined, undefined),

new LearningStep(283, undefined, undefined),

new LearningStep(284, undefined, undefined),

new LearningStep(285, undefined, undefined),

new LearningStep(286, undefined, undefined),

new LearningStep(287, undefined, undefined),

new LearningStep(288, undefined, undefined),

new LearningStep(289, undefined, undefined),

new LearningStep(290, undefined, undefined),

new LearningStep(
  290,
  "",

  new Question(
    "What value did sum(0,a) return?",
    [0,10,1],
    ["","",""] ) ),

new LearningStep(
  290,
  "**Recursion** works by defining the solution of a problem
       in terms of\n1) making small progress\n2) the
       solution to the rest of the problem\n3) how to
       combine the small progress and the rest of the
       solution\n\n
  Here's an example of a **recursive** definition of sum
       using english words. The sum of a list of numbers is
       the first number plus the sum of the remaining
       numbers.\n\n
  The small progress is *looking up the element at position i
       *. \n\n
  The solution to the rest of the problem is **making the
       recursive call to the function itself, with slightly
       different arguments**. \n\n
  How you combine the small progress and rest of the solution
        is *adding them*.",
  undefined ),

new LearningStep(
    290,
    "Our goal here is not to teach you how to design recursive
         solutions to problems, just how the computer executes
          them. We also want to give you a little insight into
          why and how they work.\n\n
    We also want to raise your interest. Recursion is most
         powerful with problems that are really hard to solve.
          It lets you build a **full** solution even when you
         can only figure out how to make the **tiniest** bit
         of progress towards a full solution.",
    undefined ),

  new LearningStep(
    290,
    "Onwards!",
    undefined )]
                    ) /* end LearningSteps */
                    ) /* end Action */,
new Action (
                    new Program(`


var y1 = 3;
var outcome1 = y1;
for (var x = y1; x <= 0; x = x-1){
    outcome1 = 0;
};


function count_down(x){
    if( x <= 0){
        return 0;
    } else {
        return count_down(x-1);
    }
};

count_down(3);


var y = 3;
var outcome = y;
for (var x = y; x >= 0; x = x-1){
    outcome = 0;
};

function count_down2(x){
    if( x >= 0){
        return 0;
    } else {
        return count_down2(x-1);
    }
};


count_down2(3);
count_down2(y);



function add_down(x){
    if( x<= 0){
        return x;
    } else {
        return x + count_down(x-1);
    }
};

add_down(5);

var y = 5;
var outcome = 0;
for (var x = y; x <= 0; x = x-1){
    outcome += x;
};



function add1(x){
    return x+1;
};

function add2(x){
   return add1(add1(x));
};
```

```
add2(0);
add2(1);

function add3(x){
    return add1(x) + add2(x);
}

add3(0);
add3(1);
```

```
            `
            ) /* end Program */,
                    new LearningSteps(
        /* Learning Steps */
        [
  new LearningStep(
    0,
    "## Function Review\n\n
    Now we will see some other examples of recursion and review
          functions. Some of these functions will behave oddly
        . It's just to check that you understand how they
        would execute.",
    undefined ),

  new LearningStep(
    0,
    "First, let's make a for loop that counts down one at a
         time until it reaches 0.",
    undefined ),

  new LearningStep(0, undefined, undefined),

  new LearningStep(1, undefined, undefined),

  new LearningStep(2, undefined, undefined),

  new LearningStep(3, undefined, undefined),

  new LearningStep(4, undefined, undefined),

  new LearningStep(5, undefined, undefined),

  new LearningStep(6, undefined, undefined),

  new LearningStep(7, undefined, undefined),

  new LearningStep(8, undefined, undefined),

  new LearningStep(9, undefined, undefined),

  new LearningStep(10, undefined, undefined),

  new LearningStep(11, undefined, undefined),

  new LearningStep(12, undefined, undefined),

  new LearningStep(13, undefined, undefined),

  new LearningStep(14, undefined, undefined),

  new LearningStep(15, undefined, undefined),

  new LearningStep(16, undefined, undefined),

  new LearningStep(17, undefined, undefined),

  new LearningStep(18, undefined, undefined),

  new LearningStep(19, undefined, undefined),

  new LearningStep(20, undefined, undefined),

  new LearningStep(
    21,
    "You can't tell what a function\ndoes just by looking at
         its name.",
    undefined ),

  new LearningStep(21, undefined, undefined),

  new LearningStep(22, undefined, undefined),

  new LearningStep(23, undefined, undefined),

  new LearningStep(24, undefined, undefined),

  new LearningStep(25, undefined, undefined),

  new LearningStep(
    26,
    "Try mentally executing this function call, then step
         through to confirm your sense of it.",
    undefined ),

  new LearningStep(27, undefined, undefined),

  new LearningStep(28, undefined, undefined),

  new LearningStep(29, undefined, undefined),

  new LearningStep(30, undefined, undefined),

  new LearningStep(31, undefined, undefined),

  new LearningStep(32, undefined, undefined),

  new LearningStep(33, undefined, undefined),

  new LearningStep(34, undefined, undefined),

  new LearningStep(35, undefined, undefined),

  new LearningStep(36, undefined, undefined),

  new LearningStep(37, undefined, undefined),

  new LearningStep(38, undefined, undefined),

  new LearningStep(39, undefined, undefined),

  new LearningStep(40, undefined, undefined),

  new LearningStep(41, undefined, undefined),

  new LearningStep(42, undefined, undefined),

  new LearningStep(43, undefined, undefined),

  new LearningStep(44, undefined, undefined),

  new LearningStep(45, undefined, undefined),

  new LearningStep(46, undefined, undefined),

  new LearningStep(47, undefined, undefined),

  new LearningStep(48, undefined, undefined),

  new LearningStep(49, undefined, undefined),

  new LearningStep(50, undefined, undefined),

  new LearningStep(51, undefined, undefined),

  new LearningStep(52, undefined, undefined),

  new LearningStep(53, undefined, undefined),

  new LearningStep(54, undefined, undefined),

  new LearningStep(55, undefined, undefined),

  new LearningStep(56, undefined, undefined),

  new LearningStep(57, undefined, undefined),

  new LearningStep(58, undefined, undefined),

  new LearningStep(59, undefined, undefined),

  new LearningStep(60, undefined, undefined),

  new LearningStep(61, undefined, undefined),
```

```
new LearningStep(62, undefined, undefined),
new LearningStep(63, undefined, undefined),
new LearningStep(64, undefined, undefined),
new LearningStep(65, undefined, undefined),
new LearningStep(66, undefined, undefined),
new LearningStep(67, undefined, undefined),
new LearningStep(68, undefined, undefined),
new LearningStep(69, undefined, undefined),
new LearningStep(70, undefined, undefined),
new LearningStep(71, undefined, undefined),
new LearningStep(72, undefined, undefined),
new LearningStep(73, undefined, undefined),
new LearningStep(74, undefined, undefined),
new LearningStep(75, undefined, undefined),
new LearningStep(76, undefined, undefined),
new LearningStep(77, undefined, undefined),
new LearningStep(78, undefined, undefined),
new LearningStep(
  78,
  undefined,
  new Question(
    "What value is on the stack?",
    [100,1,10],
    ["","",""] ) ),
new LearningStep(
  79,
  "What changes next?",
  undefined ),
new LearningStep(79, undefined, undefined),
new LearningStep(80, undefined, undefined),
new LearningStep(81, undefined, undefined),
new LearningStep(82, undefined, undefined),
new LearningStep(83, undefined, undefined),
new LearningStep(84, undefined, undefined),
new LearningStep(85, undefined, undefined),
new LearningStep(86, undefined, undefined),
new LearningStep(87, undefined, undefined),
new LearningStep(88, undefined, undefined),
new LearningStep(89, undefined, undefined),
new LearningStep(90, undefined, undefined),
new LearningStep(91, undefined, undefined),
new LearningStep(92, undefined, undefined),
new LearningStep(93, undefined, undefined),
new LearningStep(94, undefined, undefined),
new LearningStep(95, undefined, undefined),
new LearningStep(96, undefined, undefined),
new LearningStep(97, undefined, undefined),
new LearningStep(98, undefined, undefined),
new LearningStep(99, undefined, undefined),
new LearningStep(100, undefined, undefined),
new LearningStep(101, undefined, undefined),
new LearningStep(102, undefined, undefined),
new LearningStep(103, undefined, undefined),
new LearningStep(104, undefined, undefined),
new LearningStep(105, undefined, undefined),
new LearningStep(106, undefined, undefined),
new LearningStep(107, undefined, undefined),
new LearningStep(108, undefined, undefined),
new LearningStep(109, undefined, undefined),
new LearningStep(110, undefined, undefined),
new LearningStep(111, undefined, undefined),
new LearningStep(112, undefined, undefined),
new LearningStep(113, undefined, undefined),
new LearningStep(114, undefined, undefined),
new LearningStep(115, undefined, undefined),
new LearningStep(116, undefined, undefined),
new LearningStep(117, undefined, undefined),
new LearningStep(118, undefined, undefined),
new LearningStep(119, undefined, undefined),
new LearningStep(120, undefined, undefined),
new LearningStep(121, undefined, undefined),
new LearningStep(122, undefined, undefined),
new LearningStep(123, undefined, undefined),
new LearningStep(124, undefined, undefined),
new LearningStep(125, undefined, undefined),
new LearningStep(126, undefined, undefined),
new LearningStep(127, undefined, undefined),
new LearningStep(128, undefined, undefined),
new LearningStep(129, undefined, undefined),
new LearningStep(130, undefined, undefined),
new LearningStep(131, undefined, undefined),
new LearningStep(132, undefined, undefined),
new LearningStep(133, undefined, undefined),
new LearningStep(134, undefined, undefined),
new LearningStep(135, undefined, undefined),
new LearningStep(136, undefined, undefined),
new LearningStep(137, undefined, undefined),
new LearningStep(138, undefined, undefined),
new LearningStep(139, undefined, undefined),
```

```
new LearningStep(140, undefined, undefined),
new LearningStep(141, undefined, undefined),
new LearningStep(142, undefined, undefined),
new LearningStep(143, undefined, undefined),
new LearningStep(144, undefined, undefined),
new LearningStep(145, undefined, undefined),
new LearningStep(146, undefined, undefined),
new LearningStep(147, undefined, undefined),
new LearningStep(148, undefined, undefined),
new LearningStep(149, undefined, undefined),
new LearningStep(150, undefined, undefined),
new LearningStep(151, undefined, undefined),
new LearningStep(152, undefined, undefined),
new LearningStep(153, undefined, undefined),
new LearningStep(154, undefined, undefined),
new LearningStep(155, undefined, undefined),
new LearningStep(156, undefined, undefined),
new LearningStep(157, undefined, undefined),
new LearningStep(158, undefined, undefined),
new LearningStep(159, undefined, undefined),
new LearningStep(160, undefined, undefined),
new LearningStep(161, undefined, undefined),
new LearningStep(162, undefined, undefined),
new LearningStep(163, undefined, undefined),
new LearningStep(164, undefined, undefined),
new LearningStep(165, undefined, undefined),
new LearningStep(
  166,
  "Try mentally executing countdown2 - what will happen?",
  undefined ),
new LearningStep(166, undefined, undefined),
new LearningStep(167, undefined, undefined),
new LearningStep(168, undefined, undefined),
new LearningStep(169, undefined, undefined),
new LearningStep(170, undefined, undefined),
new LearningStep(171, undefined, undefined),
new LearningStep(172, undefined, undefined),
new LearningStep(173, undefined, undefined),
new LearningStep(174, undefined, undefined),
new LearningStep(175, undefined, undefined),
new LearningStep(176, undefined, undefined),
new LearningStep(177, undefined, undefined),
new LearningStep(178, undefined, undefined),
new LearningStep(179, undefined, undefined),
new LearningStep(
  179,
  undefined,
  new Question(
    "What value is on the stack?",
    [100,10,1],
    ["","",""] ) ),
new LearningStep(180, undefined, undefined),
new LearningStep(181, undefined, undefined),
new LearningStep(182, undefined, undefined),
new LearningStep(183, undefined, undefined),
new LearningStep(184, undefined, undefined),
new LearningStep(185, undefined, undefined),
new LearningStep(186, undefined, undefined),
new LearningStep(187, undefined, undefined),
new LearningStep(188, undefined, undefined),
new LearningStep(189, undefined, undefined),
new LearningStep(190, undefined, undefined),
new LearningStep(191, undefined, undefined),
new LearningStep(192, undefined, undefined),
new LearningStep(193, undefined, undefined),
new LearningStep(
  193,
  undefined,
  new Question(
    "What value is on the stack?",
    [1,100,5],
    ["","",""] ) ),
new LearningStep(
  194,
  "Try mentally executing this function.",
  undefined ),
new LearningStep(194, undefined, undefined),
new LearningStep(195, undefined, undefined),
new LearningStep(196, undefined, undefined),
new LearningStep(197, undefined, undefined),
new LearningStep(198, undefined, undefined),
new LearningStep(199, undefined, undefined),
new LearningStep(200, undefined, undefined),
new LearningStep(201, undefined, undefined),
new LearningStep(202, undefined, undefined),
new LearningStep(203, undefined, undefined),
new LearningStep(204, undefined, undefined),
new LearningStep(205, undefined, undefined),
new LearningStep(206, undefined, undefined),
new LearningStep(207, undefined, undefined),
new LearningStep(208, undefined, undefined),
new LearningStep(209, undefined, undefined),
new LearningStep(210, undefined, undefined),
```

```
new LearningStep(211, undefined, undefined),
new LearningStep(212, undefined, undefined),
new LearningStep(213, undefined, undefined),
new LearningStep(214, undefined, undefined),
new LearningStep(215, undefined, undefined),
new LearningStep(216, undefined, undefined),
new LearningStep(217, undefined, undefined),
new LearningStep(218, undefined, undefined),
new LearningStep(219, undefined, undefined),
new LearningStep(220, undefined, undefined),
new LearningStep(221, undefined, undefined),
new LearningStep(222, undefined, undefined),
new LearningStep(223, undefined, undefined),
new LearningStep(224, undefined, undefined),
new LearningStep(225, undefined, undefined),
new LearningStep(226, undefined, undefined),
new LearningStep(227, undefined, undefined),
new LearningStep(228, undefined, undefined),
new LearningStep(229, undefined, undefined),
new LearningStep(230, undefined, undefined),
new LearningStep(231, undefined, undefined),
new LearningStep(232, undefined, undefined),
new LearningStep(233, undefined, undefined),
new LearningStep(234, undefined, undefined),
new LearningStep(235, undefined, undefined),
new LearningStep(236, undefined, undefined),
new LearningStep(237, undefined, undefined),
new LearningStep(238, undefined, undefined),
new LearningStep(239, undefined, undefined),
new LearningStep(240, undefined, undefined),
new LearningStep(241, undefined, undefined),
new LearningStep(242, undefined, undefined),
new LearningStep(243, undefined, undefined),
new LearningStep(244, undefined, undefined),
new LearningStep(245, undefined, undefined),
new LearningStep(246, undefined, undefined),
new LearningStep(247, undefined, undefined),
new LearningStep(248, undefined, undefined),
new LearningStep(249, undefined, undefined),
new LearningStep(250, undefined, undefined),
new LearningStep(251, undefined, undefined),
new LearningStep(252, undefined, undefined),
new LearningStep(253, undefined, undefined),
new LearningStep(254, undefined, undefined),
new LearningStep(255, undefined, undefined),
new LearningStep(256, undefined, undefined),
new LearningStep(257, undefined, undefined),
new LearningStep(258, undefined, undefined),
new LearningStep(259, undefined, undefined),
new LearningStep(260, undefined, undefined),
new LearningStep(261, undefined, undefined),
new LearningStep(262, undefined, undefined),
new LearningStep(263, undefined, undefined),
new LearningStep(264, undefined, undefined),
new LearningStep(265, undefined, undefined),
new LearningStep(266, undefined, undefined),
new LearningStep(267, undefined, undefined),
new LearningStep(268, undefined, undefined),
new LearningStep(269, undefined, undefined),
new LearningStep(270, undefined, undefined),
new LearningStep(271, undefined, undefined),
new LearningStep(272, undefined, undefined),
new LearningStep(273, undefined, undefined),
new LearningStep(274, undefined, undefined),
new LearningStep(275, undefined, undefined),
new LearningStep(276, undefined, undefined),
new LearningStep(277, undefined, undefined),
new LearningStep(278, undefined, undefined),
new LearningStep(279, undefined, undefined),
new LearningStep(280, undefined, undefined),
new LearningStep(281, undefined, undefined),
new LearningStep(282, undefined, undefined),
new LearningStep(283, undefined, undefined),
new LearningStep(284, undefined, undefined),
new LearningStep(285, undefined, undefined),
new LearningStep(286, undefined, undefined),
new LearningStep(287, undefined, undefined),
new LearningStep(288, undefined, undefined),
new LearningStep(289, undefined, undefined),
new LearningStep(290, undefined, undefined),
new LearningStep(291, undefined, undefined),
new LearningStep(292, undefined, undefined),
new LearningStep(293, undefined, undefined),
new LearningStep(294, undefined, undefined),
new LearningStep(295, undefined, undefined),
```

```
new LearningStep(296, undefined, undefined),
new LearningStep(297, undefined, undefined),
new LearningStep(298, undefined, undefined),
new LearningStep(299, undefined, undefined),
new LearningStep(300, undefined, undefined),
new LearningStep(301, undefined, undefined),
new LearningStep(302, undefined, undefined),
new LearningStep(303, undefined, undefined),
new LearningStep(304, undefined, undefined),
new LearningStep(305, undefined, undefined),
new LearningStep(306, undefined, undefined),
new LearningStep(307, undefined, undefined),
new LearningStep(308, undefined, undefined),
new LearningStep(309, undefined, undefined),
new LearningStep(310, undefined, undefined),
new LearningStep(311, undefined, undefined),
new LearningStep(312, undefined, undefined),
new LearningStep(313, undefined, undefined),
new LearningStep(314, undefined, undefined),
new LearningStep(315, undefined, undefined),
new LearningStep(316, undefined, undefined),
new LearningStep(317, undefined, undefined),
new LearningStep(318, undefined, undefined),
new LearningStep(319, undefined, undefined),
new LearningStep(320, undefined, undefined),
new LearningStep(321, undefined, undefined),
new LearningStep(322, undefined, undefined),
new LearningStep(323, undefined, undefined),
new LearningStep(324, undefined, undefined),
new LearningStep(325, undefined, undefined),
new LearningStep(326, undefined, undefined),
new LearningStep(327, undefined, undefined),
new LearningStep(328, undefined, undefined),
new LearningStep(329, undefined, undefined),
new LearningStep(330, undefined, undefined),
new LearningStep(331, undefined, undefined),
new LearningStep(332, undefined, undefined),
new LearningStep(333, undefined, undefined),
new LearningStep(
  333,
  undefined,

  new Question(
    "What value is on the stack?",
    [1,5,10],
    ["","",""] ) ),
new LearningStep(334, undefined, undefined),
new LearningStep(335, undefined, undefined),
new LearningStep(336, undefined, undefined),
new LearningStep(337, undefined, undefined),
new LearningStep(338, undefined, undefined),
new LearningStep(339, undefined, undefined),
new LearningStep(340, undefined, undefined),
new LearningStep(341, undefined, undefined),
new LearningStep(342, undefined, undefined),
new LearningStep(343, undefined, undefined),
new LearningStep(344, undefined, undefined),
new LearningStep(345, undefined, undefined),
new LearningStep(346, undefined, undefined),
new LearningStep(347, undefined, undefined),
new LearningStep(348, undefined, undefined),
new LearningStep(349, undefined, undefined),
new LearningStep(350, undefined, undefined),
new LearningStep(351, undefined, undefined),
new LearningStep(352, undefined, undefined),
new LearningStep(353, undefined, undefined),
new LearningStep(354, undefined, undefined),
new LearningStep(355, undefined, undefined),
new LearningStep(356, undefined, undefined),
new LearningStep(357, undefined, undefined),
new LearningStep(358, undefined, undefined),
new LearningStep(359, undefined, undefined),
new LearningStep(360, undefined, undefined),
new LearningStep(
  360,
  undefined,

  new Question(
    "What value is on the stack?",
    [0,100,10],
    ["","",""] ) ),
new LearningStep(361, undefined, undefined),
new LearningStep(362, undefined, undefined),
new LearningStep(363, undefined, undefined),
new LearningStep(364, undefined, undefined),
new LearningStep(365, undefined, undefined),
new LearningStep(366, undefined, undefined),
new LearningStep(367, undefined, undefined),
new LearningStep(368, undefined, undefined),
new LearningStep(369, undefined, undefined),
new LearningStep(370, undefined, undefined),
new LearningStep(371, undefined, undefined),
```

```
new LearningStep(372, undefined, undefined),
new LearningStep(373, undefined, undefined),
new LearningStep(374, undefined, undefined),
new LearningStep(375, undefined, undefined),
new LearningStep(376, undefined, undefined),
new LearningStep(377, undefined, undefined),
new LearningStep(378, undefined, undefined),
new LearningStep(379, undefined, undefined),
new LearningStep(380, undefined, undefined),
new LearningStep(381, undefined, undefined),
new LearningStep(382, undefined, undefined),
new LearningStep(383, undefined, undefined),
new LearningStep(384, undefined, undefined),
new LearningStep(385, undefined, undefined),
new LearningStep(386, undefined, undefined),
new LearningStep(387, undefined, undefined),
new LearningStep(388, undefined, undefined),
new LearningStep(389, undefined, undefined),
new LearningStep(390, undefined, undefined),
new LearningStep(391, undefined, undefined),
new LearningStep(392, undefined, undefined),
new LearningStep(393, undefined, undefined),
new LearningStep(394, undefined, undefined),
new LearningStep(395, undefined, undefined),
new LearningStep(396, undefined, undefined),
new LearningStep(397, undefined, undefined),
new LearningStep(398, undefined, undefined),
new LearningStep(399, undefined, undefined),
new LearningStep(400, undefined, undefined),
new LearningStep(401, undefined, undefined),
new LearningStep(402, undefined, undefined),
new LearningStep(403, undefined, undefined),
new LearningStep(404, undefined, undefined),
new LearningStep(405, undefined, undefined),
new LearningStep(406, undefined, undefined),
new LearningStep(
  406,
  undefined,

  new Question(
    "What value is on the stack?",
    [10,0,5],
    ["","",""] ) ),
new LearningStep(407, undefined, undefined),
new LearningStep(408, undefined, undefined),
new LearningStep(409, undefined, undefined),
new LearningStep(410, undefined, undefined),
new LearningStep(411, undefined, undefined),
new LearningStep(412, undefined, undefined),
new LearningStep(413, undefined, undefined),
new LearningStep(414, undefined, undefined),
new LearningStep(415, undefined, undefined),
new LearningStep(416, undefined, undefined),
new LearningStep(417, undefined, undefined),
new LearningStep(418, undefined, undefined),
new LearningStep(419, undefined, undefined),
new LearningStep(420, undefined, undefined),
new LearningStep(421, undefined, undefined),
new LearningStep(422, undefined, undefined),
new LearningStep(423, undefined, undefined),
new LearningStep(424, undefined, undefined),
new LearningStep(425, undefined, undefined),
new LearningStep(426, undefined, undefined),
new LearningStep(427, undefined, undefined),
new LearningStep(428, undefined, undefined),
new LearningStep(429, undefined, undefined),
new LearningStep(430, undefined, undefined),
new LearningStep(431, undefined, undefined),
new LearningStep(432, undefined, undefined),
new LearningStep(433, undefined, undefined),
new LearningStep(434, undefined, undefined),
new LearningStep(435, undefined, undefined),
new LearningStep(436, undefined, undefined),
new LearningStep(437, undefined, undefined),
new LearningStep(438, undefined, undefined),
new LearningStep(439, undefined, undefined),
new LearningStep(440, undefined, undefined),
new LearningStep(441, undefined, undefined),
new LearningStep(442, undefined, undefined),
new LearningStep(443, undefined, undefined),
new LearningStep(444, undefined, undefined),
new LearningStep(445, undefined, undefined),
new LearningStep(446, undefined, undefined),
new LearningStep(447, undefined, undefined),
new LearningStep(448, undefined, undefined),
new LearningStep(449, undefined, undefined),
new LearningStep(450, undefined, undefined),
new LearningStep(451, undefined, undefined),
```

```
  new LearningStep(
    451,
    undefined,

    new Question(
      "What value is on the stack?",
      [100,0,4],
      ["","","This is a very tricky question. The add2 function
            adds 1 to x, getting 2, then add2 returns 3."] ) )
          ,

  new LearningStep(
    451,
    "Sometimes the names of functions aren't accurate and you
         have to think how the computer will execute them.",
    undefined ),

  new LearningStep(
    451,
    "Great! Onwards!",
    undefined )]
                    ) /* end LearningSteps */
                    ) /* end Action */]),
new KnowledgeUnit("Congratulations",
           "You did it!",
           [
new Action (
                    new Program(`



           `
           ) /* end Program */,
                    new LearningSteps(
       /* Learning Steps */
       [
  new LearningStep(
    0,
    "## Congratulations!\n\n
    You've finished the tutor! Wow! Think back to when you
         started learning programming and how much you've
         grown since then!\n\n
    Though there will always be struggles, with work and effort
          there is no limit to what you can achieve! Whether
         your journey is just beginning or you're farther on
         your way, we wish you the very best!\n\n
    Onwards!\n![Image]( http://www.greglnelson.info/shuttle.jpg
         )\n\n
    ",
    undefined )]
                    ) /* end LearningSteps */
                    ) /* end Action */]))
```

Appendix C

SCAFFOLDED ASSESSMENT SHOWING PARTIAL STATE INFORMATION AND EXECUTION

This is an example sequence showing program execution while showing partial state information, with covered up parts represented as ?.

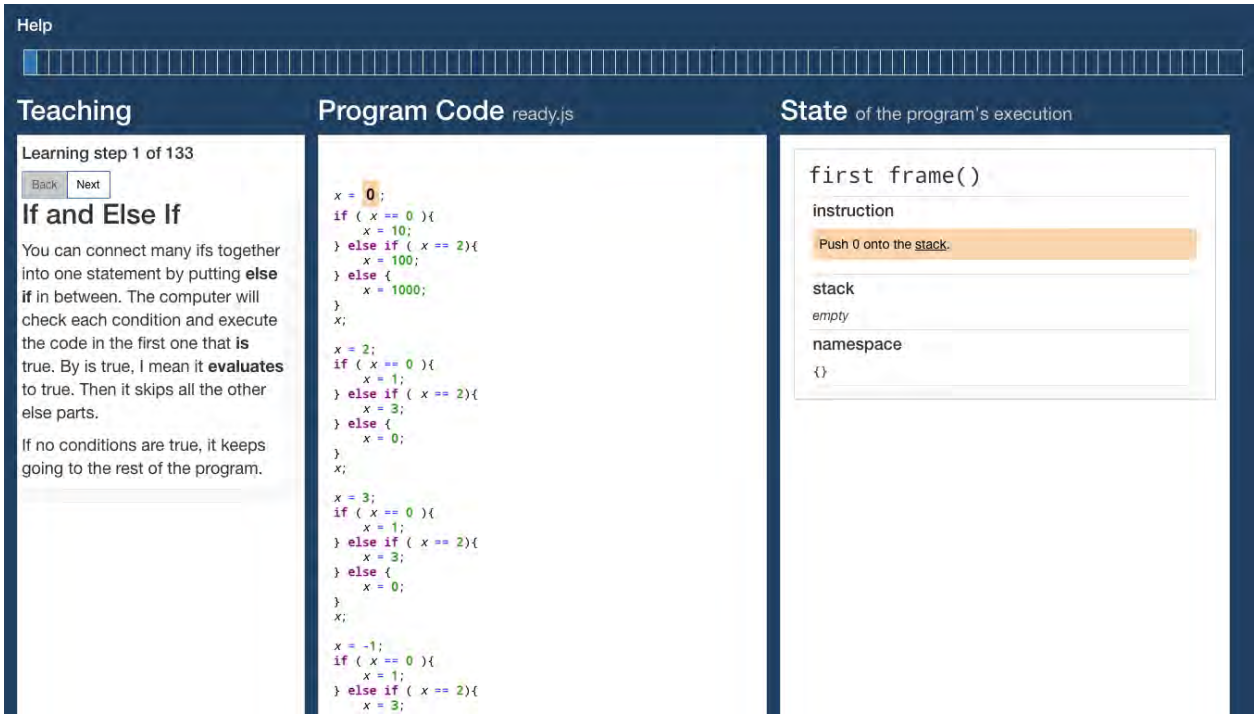


Figure C.1: Initial learning step of the first `if-else` lesson.

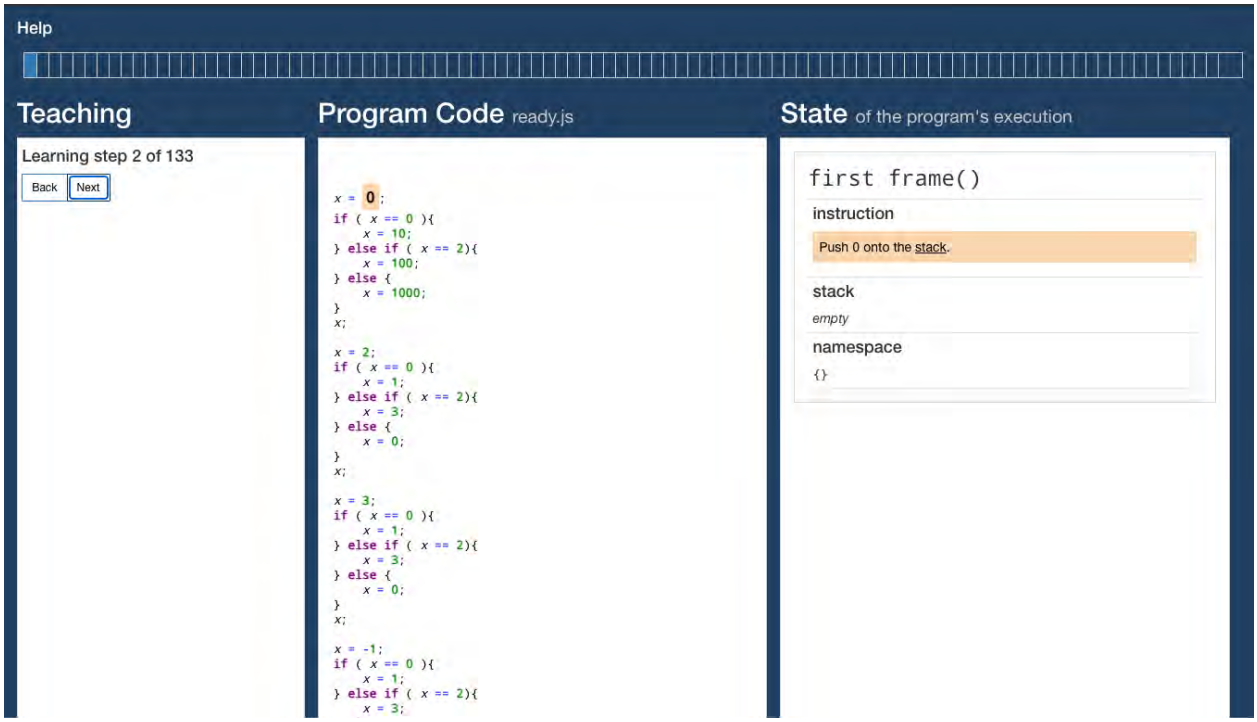


Figure C.2: Learning step 2.

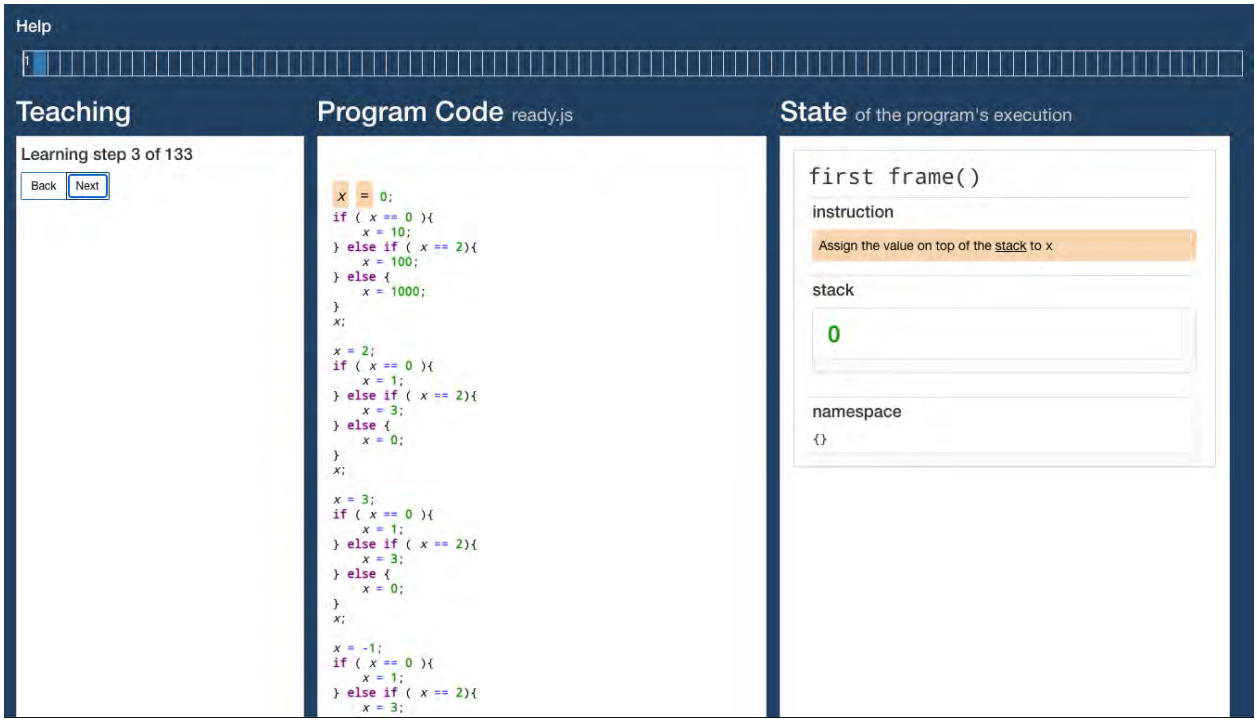


Figure C.3: Learning step 3.

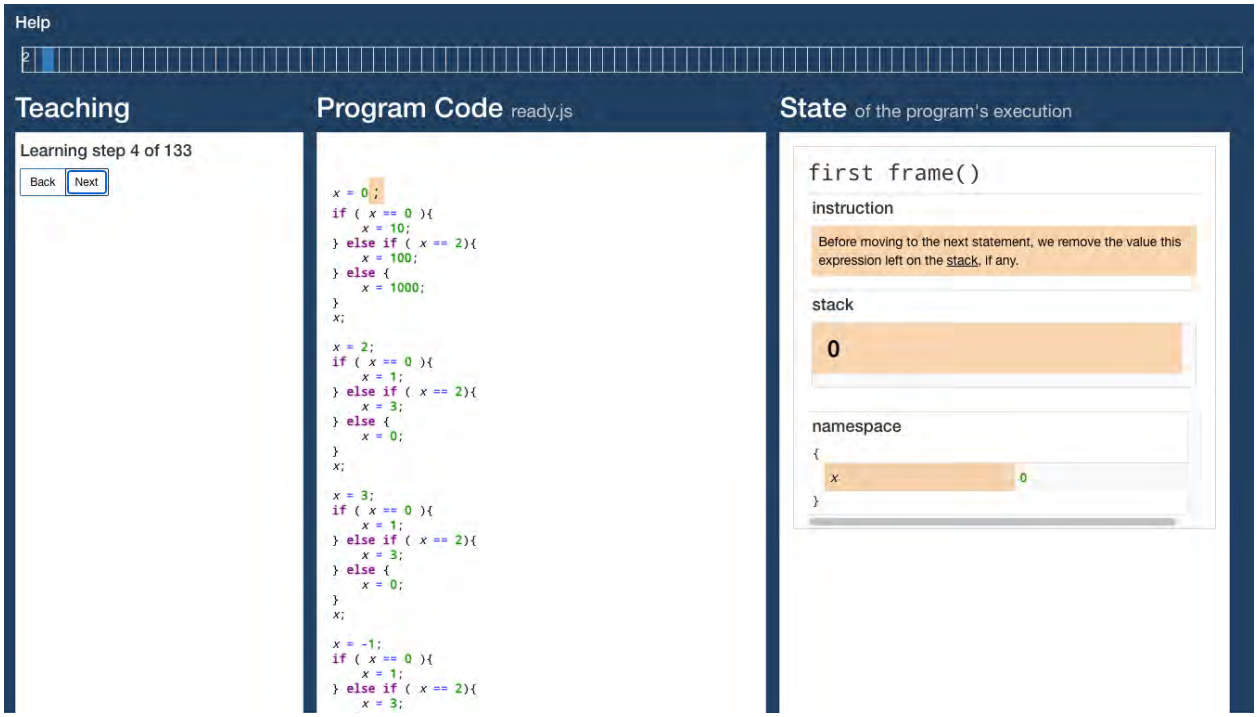


Figure C.4: Learning step 4.

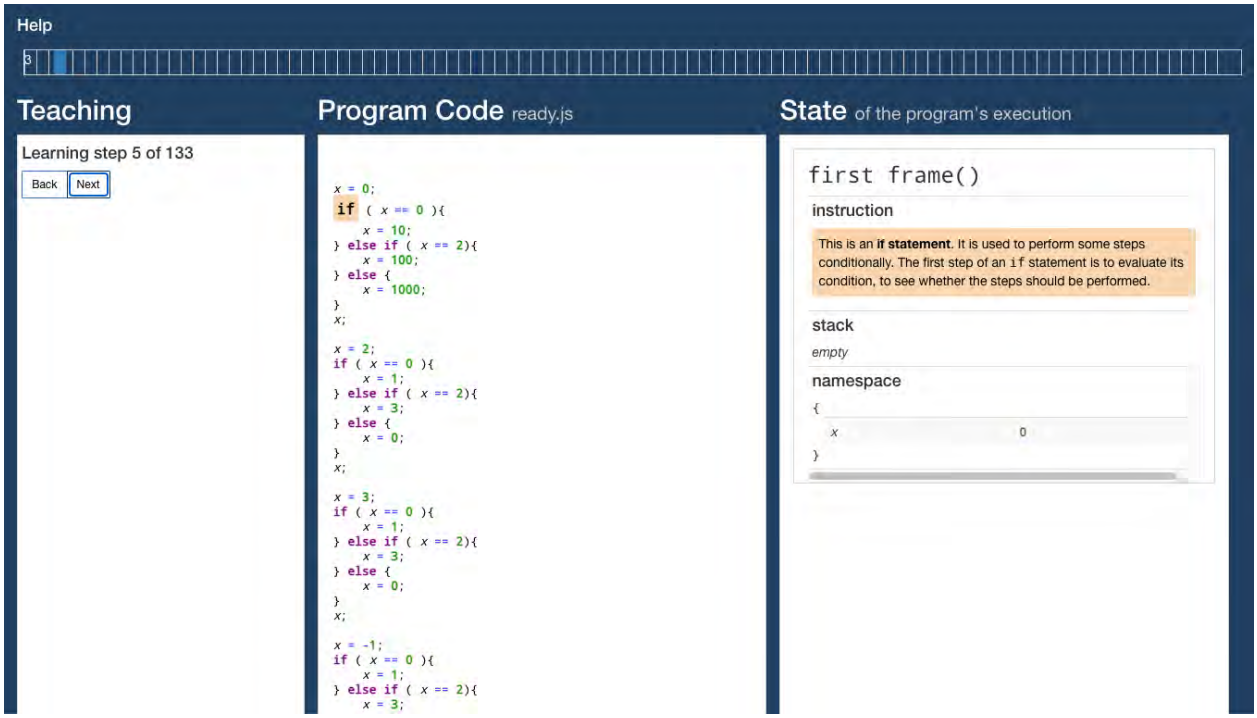


Figure C.5: Learning step 5.

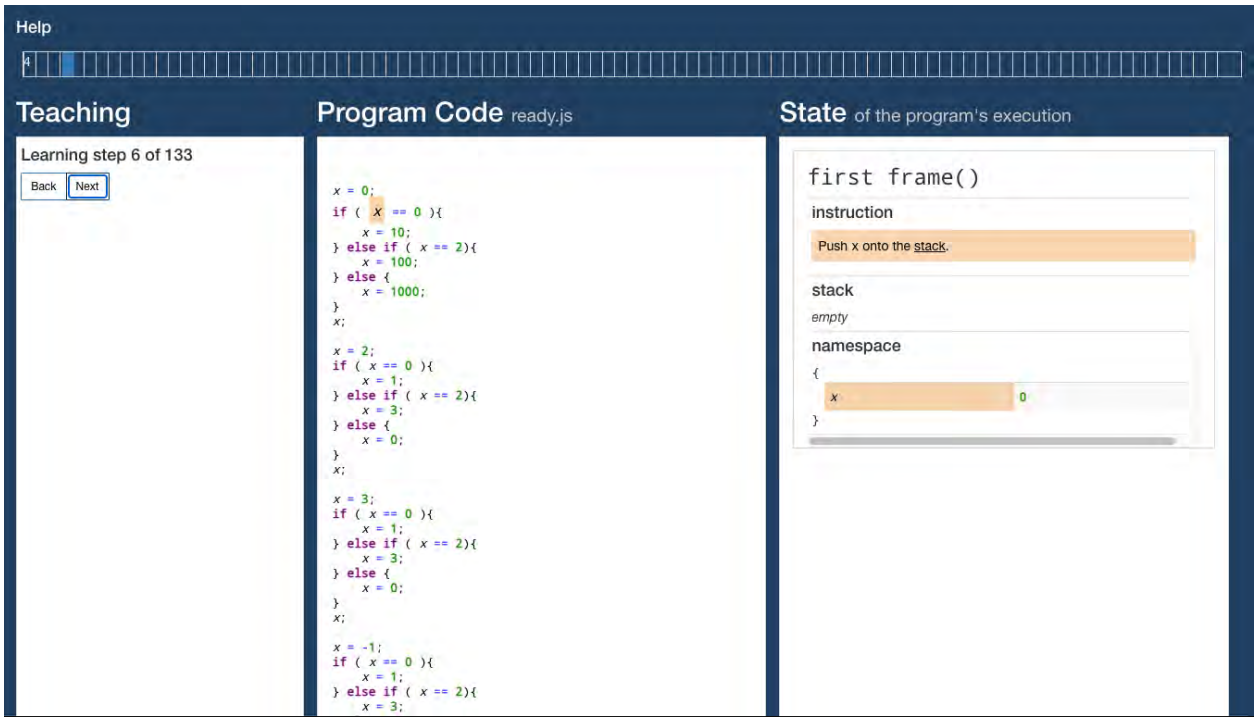


Figure C.6: Learning step 6.

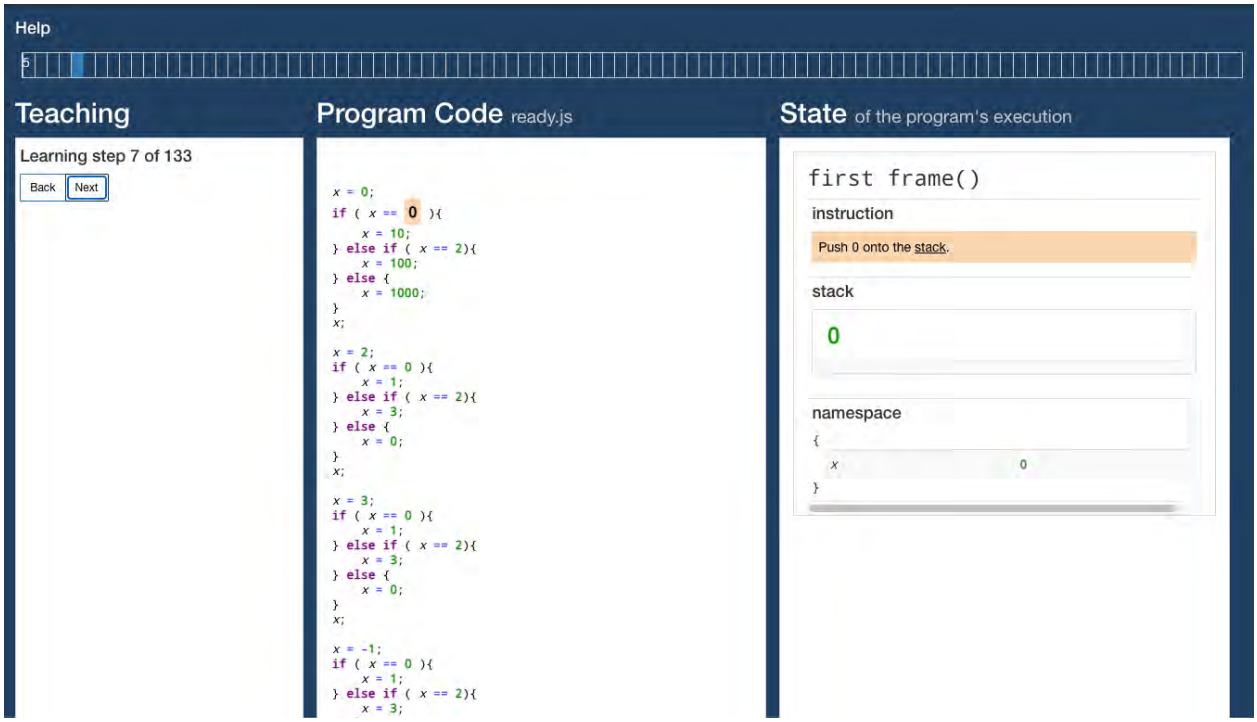


Figure C.7: Learning step 7.

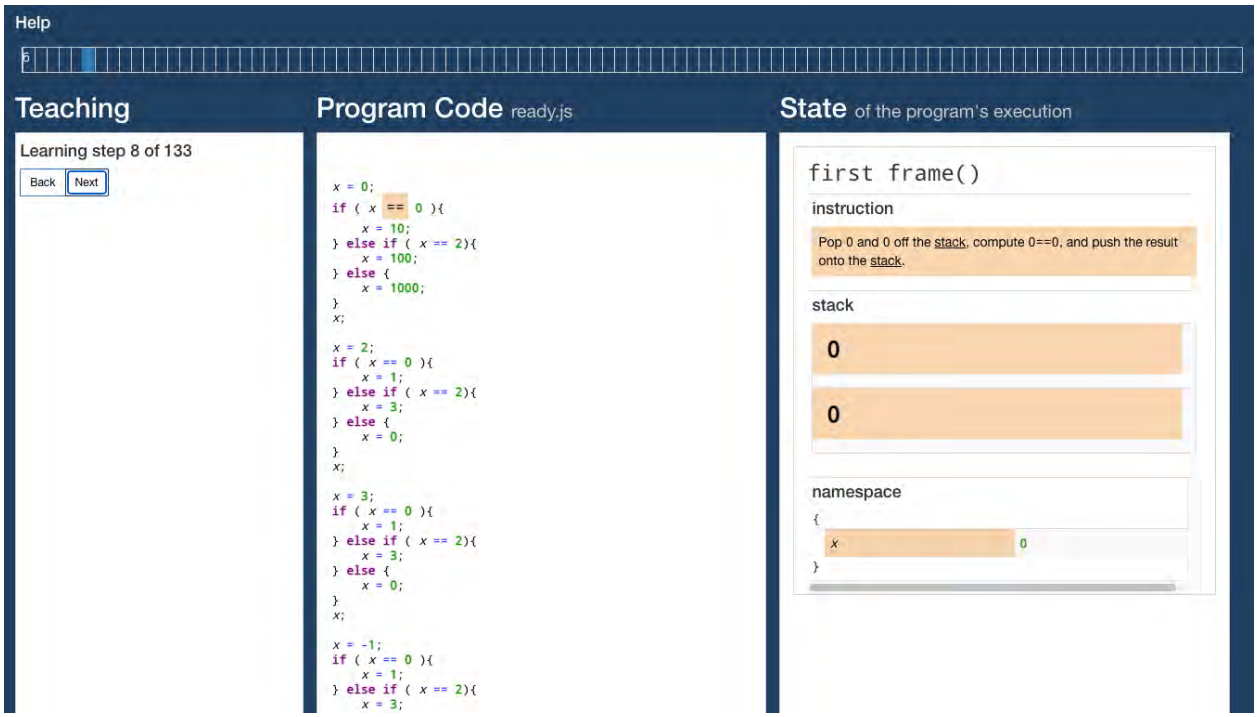


Figure C.8: Learning step 8.

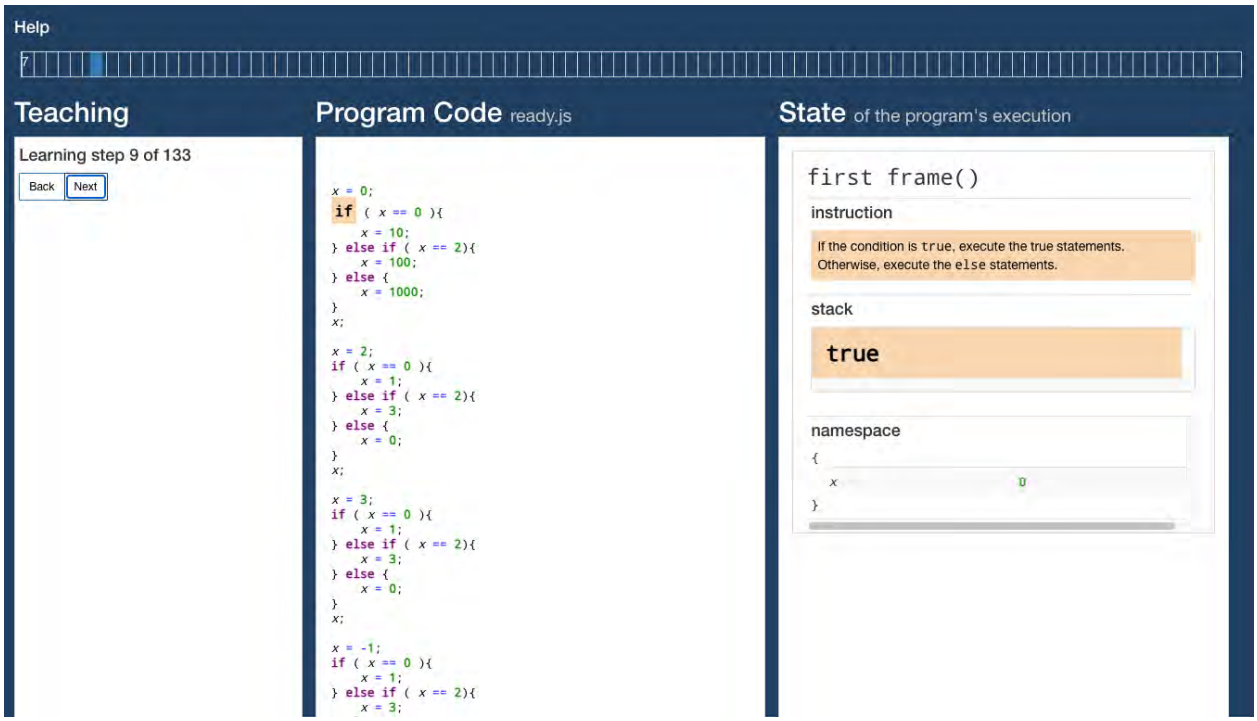


Figure C.9: Learning step 9.

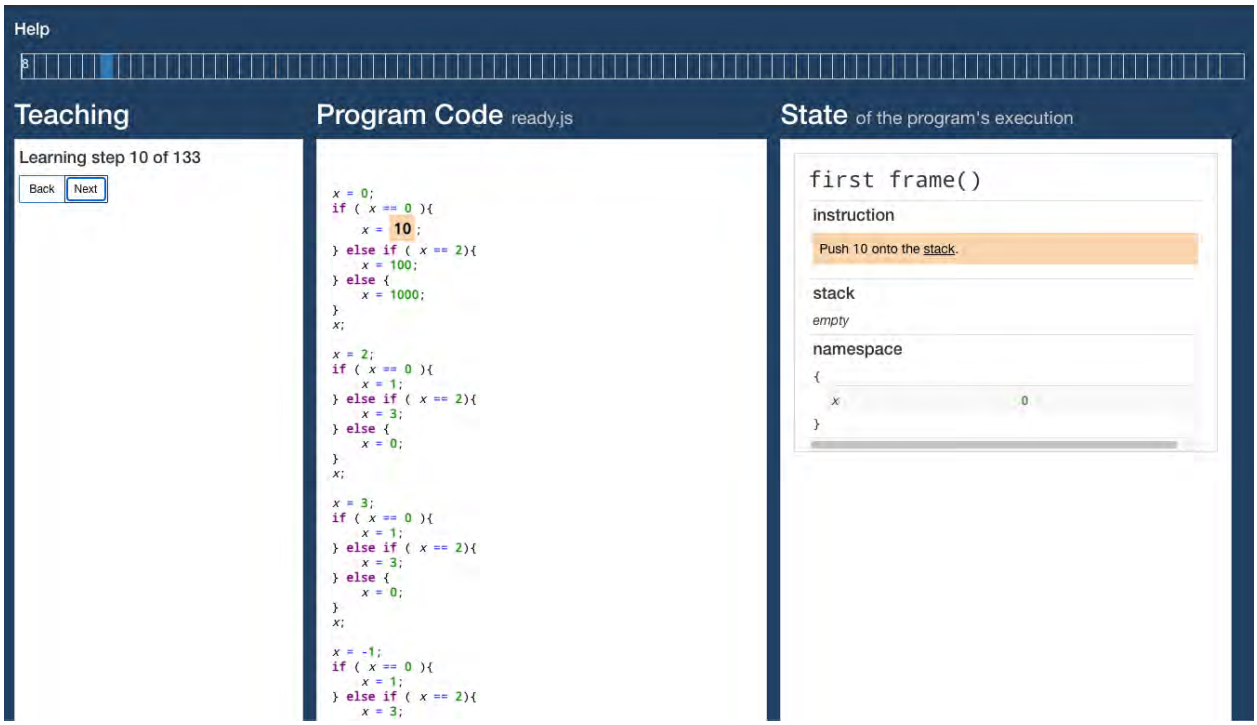


Figure C.10: Learning step 10.

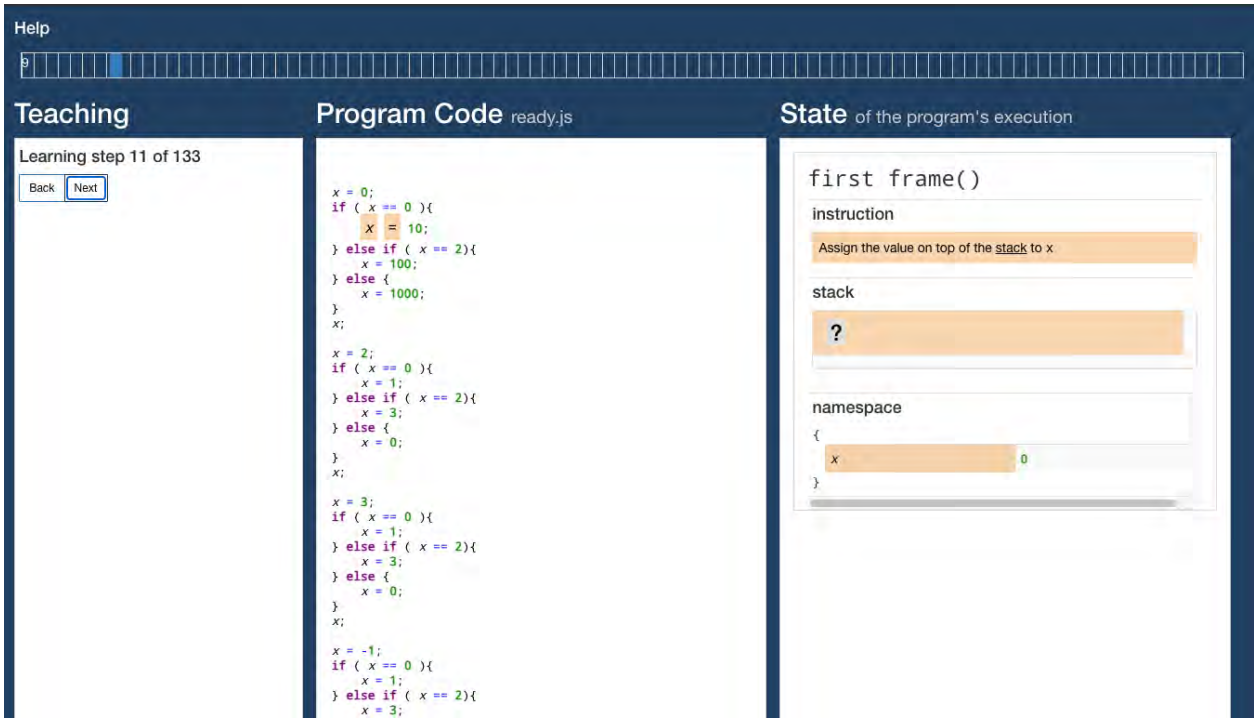


Figure C.11: Learning step 11.

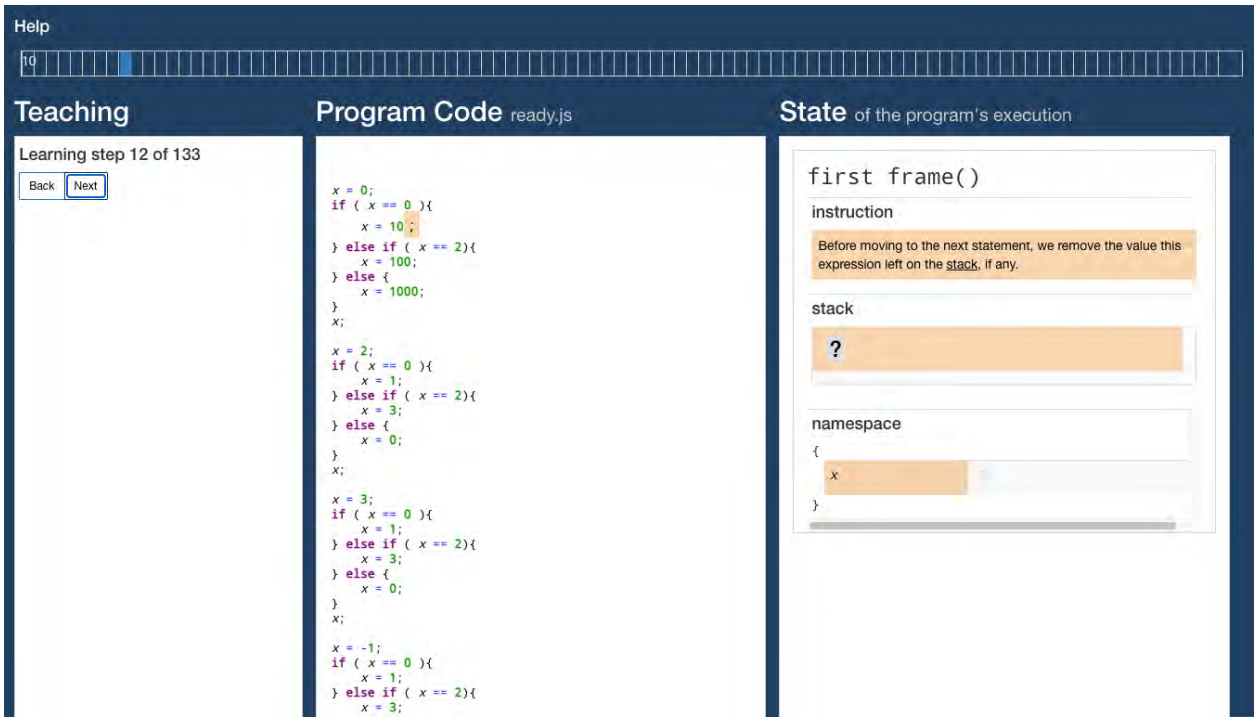


Figure C.12: Learning step 12.

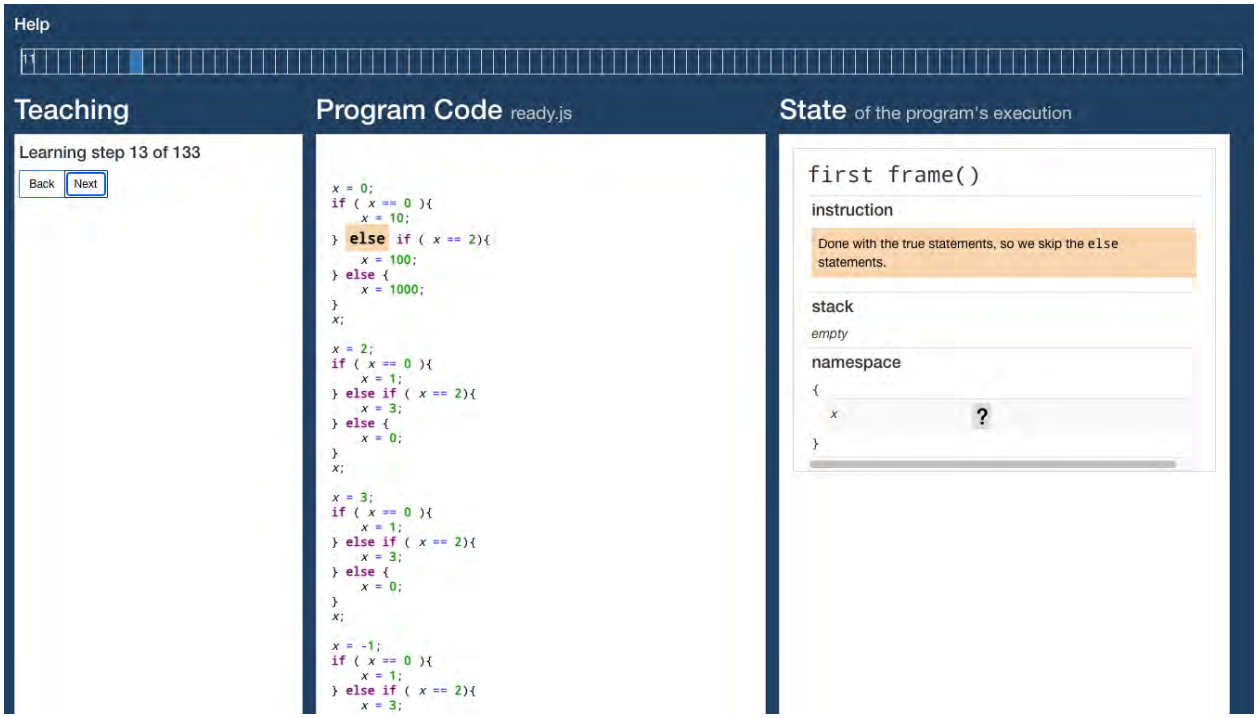


Figure C.13: Learning step 13.

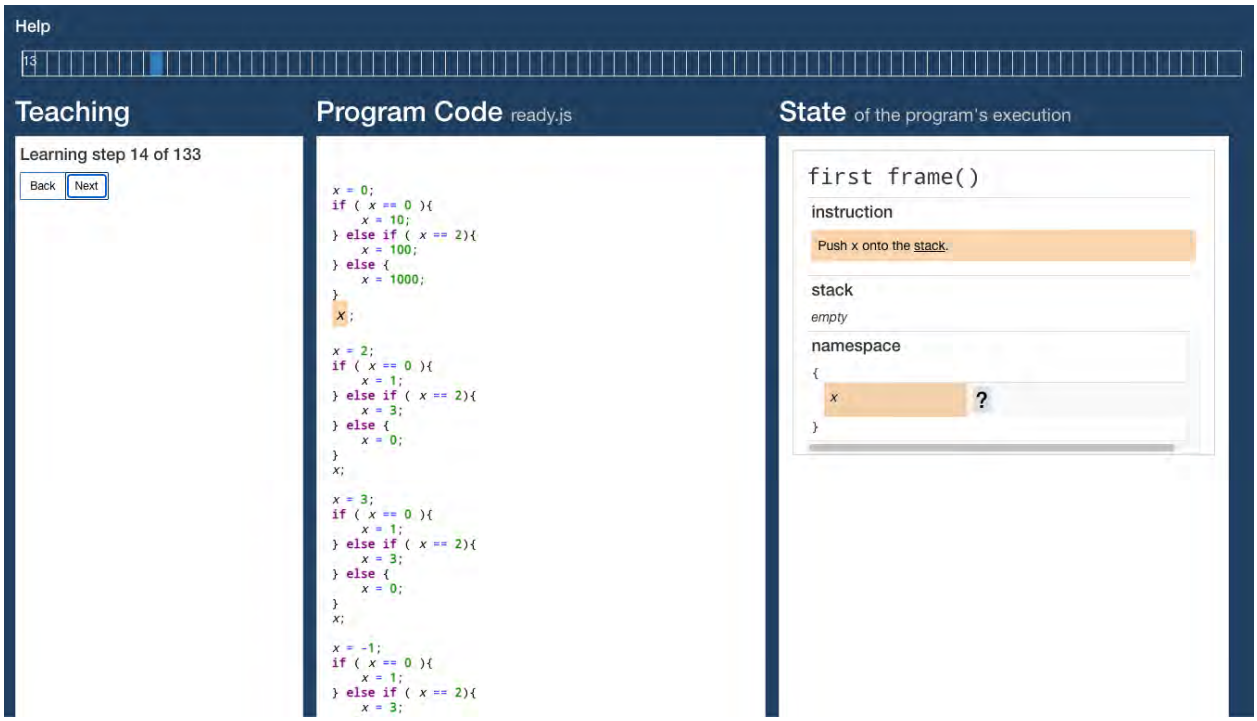


Figure C.14: Learning step 14.

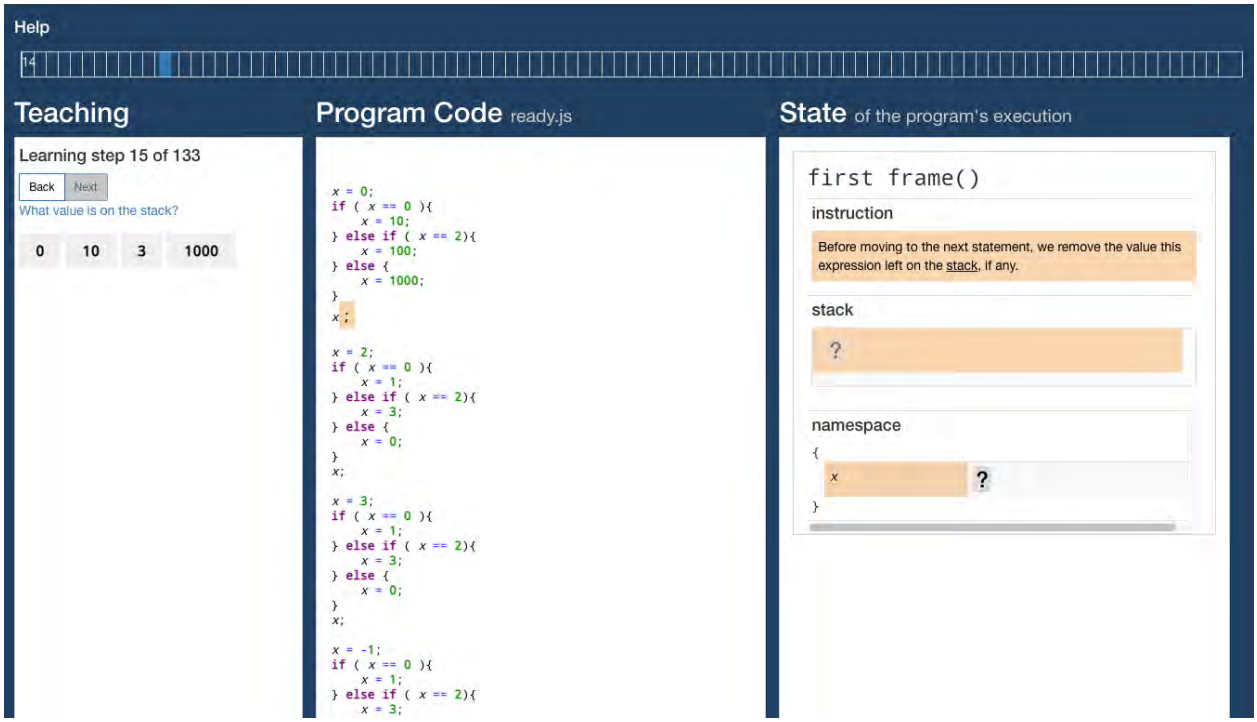


Figure C.15: Learning step 15.

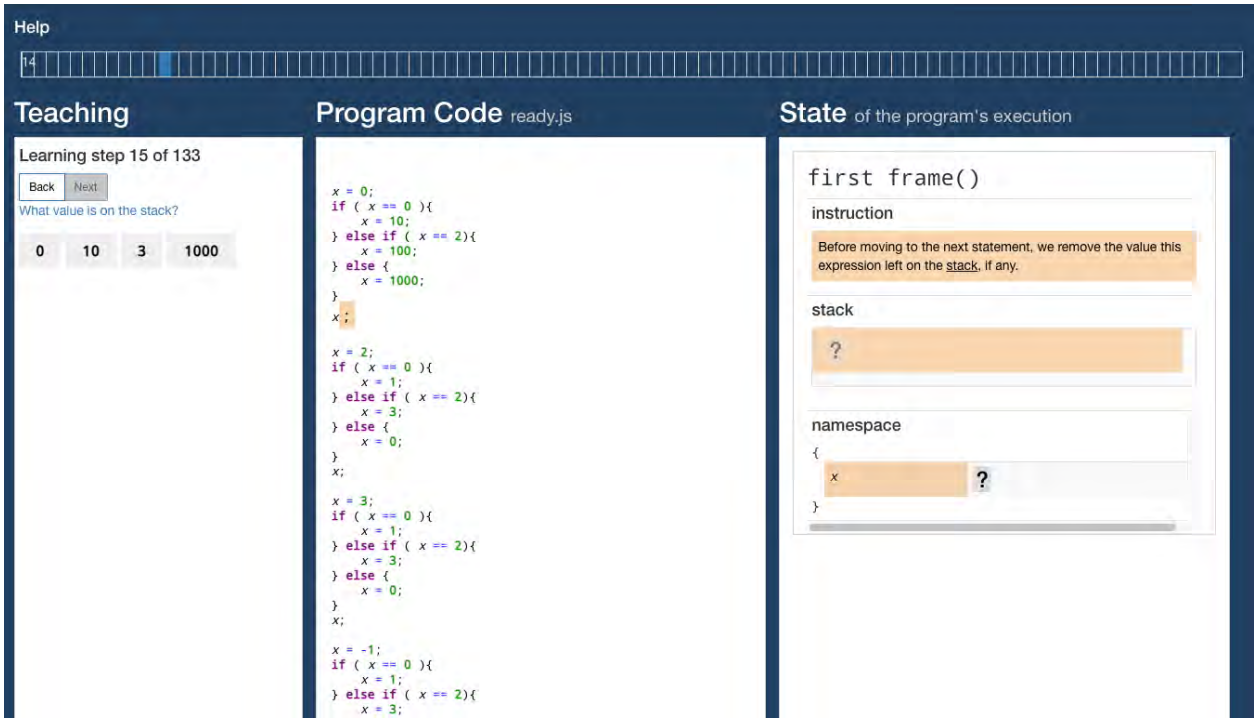


Figure C.16: Learning step 16.

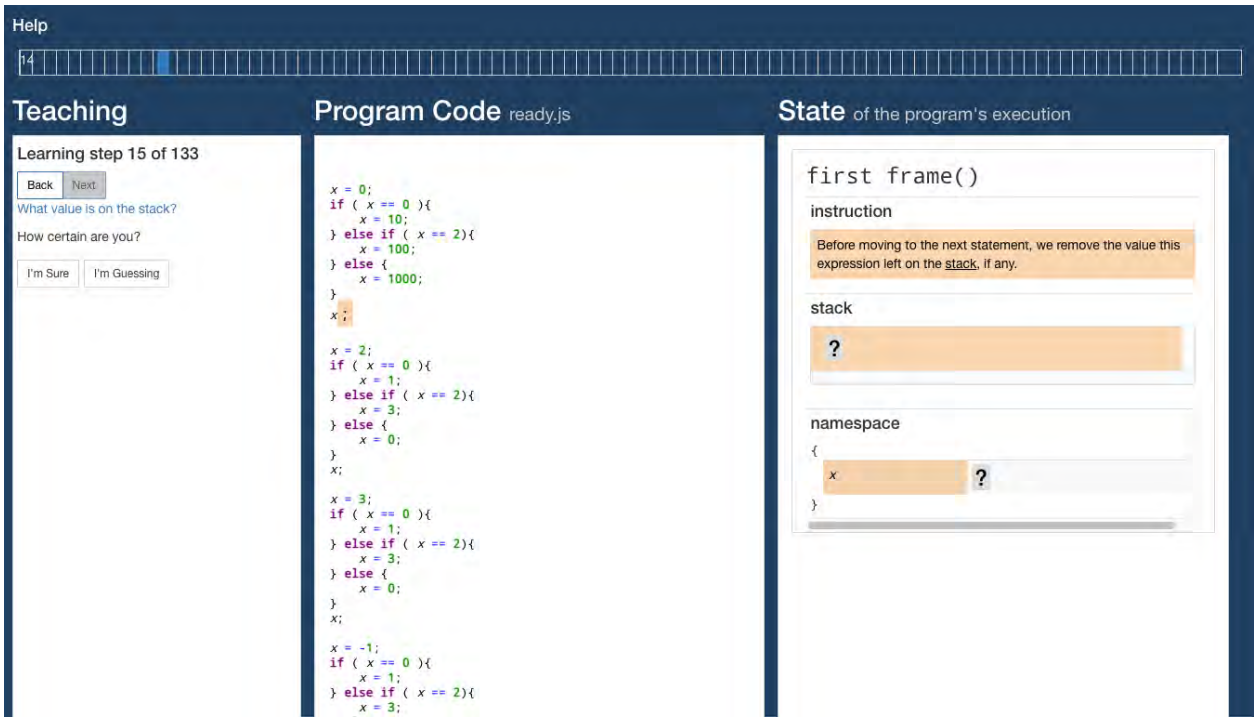


Figure C.17: The user clicked on 10, which then shows guessing or sure buttons.

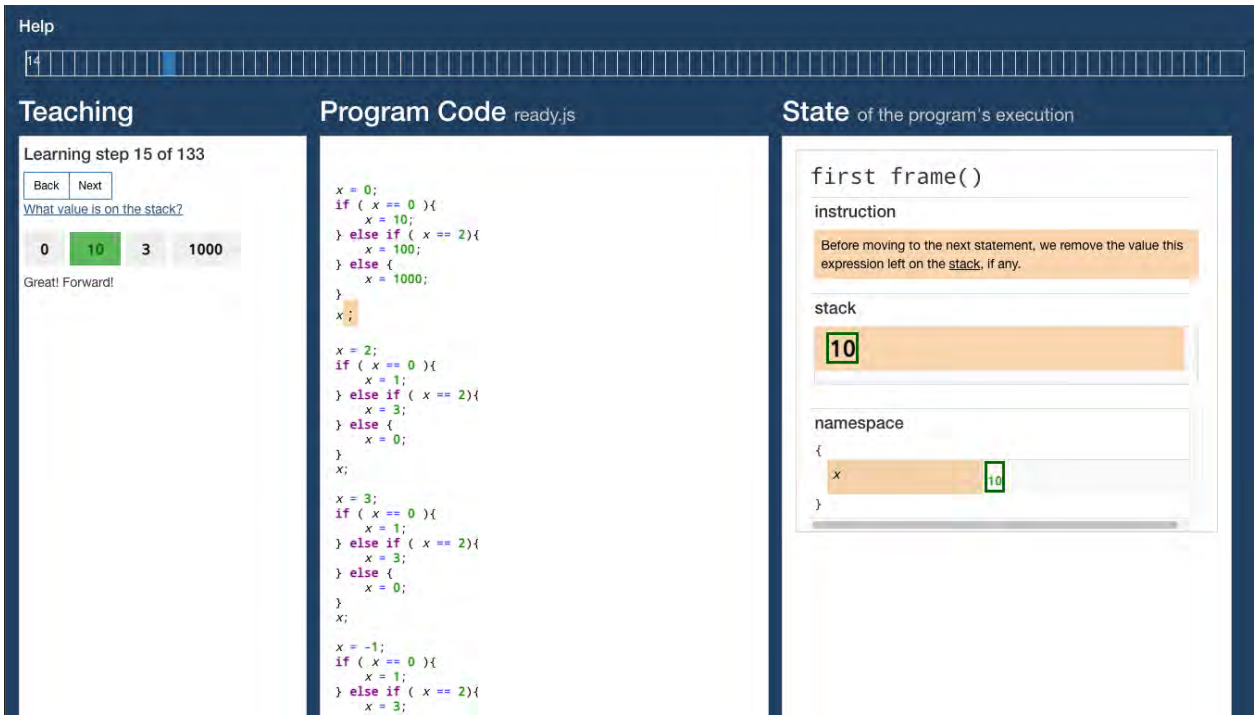


Figure C.18: The answer was correct so the value is revealed and the learner can continue to step forward through the learning steps. Otherwise a hard-coded misconception feedback string for this question by the curriculum designer would be shown, with only the most recent answer they selected highlighted, and the user could try again.

Appendix D

DOCUMENTS USED IN FORMATIVE ASSESSMENT STUDY

This appendix includes the cover sheet of the tracing assessment, the example problems used when teaching the tracing strategy, and the study protocol.

Welcome to the Program Tracing Assessment!

The purpose of this assessment is to help you focus your learning on parts you are less strong with at the moment. Please give your best effort, work carefully, **and use the explicit tracing strategy we will show you on the next page.**

All of the questions have the same format and instructions. Execute the code and enter what values are in the variables when the program finishes. The answer boxes are below each program.

This assessment has been checked and double-checked - there are no typos or mistakes in any question. Execute the exact code in the question, without making any changes.

Whatever you know right now about programming, you can learn more with more practice. Your performance on these questions has NO RELATIONSHIP with your ability to learn programming.

Turn the page to learn the explicit code tracing strategy, then continue on to the rest of the assessment.

STRATEGY: Understanding the Problem; Run the Code (like a computer).

UNDERSTAND THE PROBLEM

1. **Find where the program begins executing**. At the start of that line, draw an arrow: ⟶

RUN THE CODE

2. Execute each line according to the rules of Javascript (similar to Java, Python, C, C++, R, and other procedural programming languages):
    a. From the syntax, determine the rule for each part of the line.
    b. Follow the rules.
    c. Update memory table(s) *(here is an example memory table* ⟶
    d. Find the code for the next part.
    e. Repeat until the program terminates.

Function Name:	
Name	Value
Return	

If you need more memory table sheets, you can ask the person administering the test for more without any penalty.

For all the questions, please follow these instructions:

If you randomly guess an answer, it is harder to figure out what you know. However, you also might really know but lack confidence in your knowledge. **To help us understand your current knowledge, please fill one or more of the boxes under each answer** - these are

- “I made a random guess”
- “I made an educated / informed guess.”
- “I am reasonably confident my answer is correct.”

How many windows are there in the place you currently live?

Name 12 animals.

Name 10 places people live.

Execute the code and enter what values are in the variables when the program finishes. Fill in the answer boxes on this page (at the bottom) - you should copy the values from your memory table sheet.

```
var x = 2;
x = 5;
var y = 10;
y = 2;
```

Function Name:	
Name	Value
Return	

Function Name:	
Name	Value
Return	

Your Answers:

x is _____

y is _____

Function Name:	
Name	Value
Return	

Execute the code and enter what values are in the variables when the program finishes. Fill in the answer boxes on this page (at the bottom) - you should copy the values from your memory table sheet.

```
var x = 2;
x = 5;
var y = 10;
y = 2;
```

Function Name:	
Name	Value
Return	

Function Name:	
Name	Value
Return	

Your Answers:

x is _____

y is _____

Function Name:	
Name	Value
Return	

Execute the code and enter what values are in the variables when the program finishes. Fill in the answer boxes on this page (at the bottom) - you should copy the values from your memory table sheet.

```
var x = 5;
x = 10;
var y = 6;
x = y;
```

Function Name:	
Name	Value
Return	

Function Name:	
Name	Value
Return	

Your Answers:

x is _____

y is _____

Function Name:	
Name	Value
Return	

Execute the code and enter what values are in the variables when the program finishes. Fill in the answer boxes on this page (at the bottom) - you should copy the values from your memory table sheet.

```
var x = 5;
x = 10;
var y = 6;
x = y;
```

Function Name:	
Name	Value
Return	

Function Name:	
Name	Value
Return	

Your Answers:

x is _____

y is _____

Function Name:	
Name	Value
Return	

Execute the code and enter what values are in the variables when the program finishes. Fill in the answer boxes on this page (at the bottom) - you should copy the values from your memory table sheet.

```
var x = 5;
var y = 9;
function f() {
  return 2;
}
y = f();
```

Function Name:	
Name	Value
Return	

Function Name:	
Name	Value
Return	

Function Name:	
Name	Value
Return	

Your Answers:

x is _____

y is _____

Execute the code and enter what values are in the variables when the program finishes. Fill in the answer boxes on this page (at the bottom) - you should copy the values from your memory table sheet.

```
var x = 5;
var y = 9;
function f() {
  return 2;
}
y = f();
```

Function Name:	
Name	Value
Return	

Function Name:	
Name	Value
Return	

Function Name:	
Name	Value
Return	

Your Answers:

x is _____

y is _____

Execute the code and enter what values are in the variables when the program finishes. Fill in the answer boxes on this page (at the bottom) - you should copy the values from your memory table sheet.

```
var x = 1;
var y = 6;
function f(c) {
  return 3;
}
y = f(2);
```

Function Name:	
Name	Value
Return	

Function Name:	
Name	Value
Return	

Your Answers:

x is _____

y is _____

Function Name:	
Name	Value
Return	

Execute the code and enter what values are in the variables when the program finishes. Fill in the answer boxes on this page (at the bottom) - you should copy the values from your memory table sheet.

```
var x = 1;
var y = 6;
function f(c) {
  return 3;
}
y = f(2);
```

Function Name:	
Name	Value
Return	

Function Name:	
Name	Value
Return	

Your Answers:

x is _____

y is _____

Function Name:	
Name	Value
Return	

Study Date: _____ Real Start Time: _____ Location: _____ Strategy: Yes / No

Participant's First Name + Last Initial: _____ Interviewer: _____

Protocol

1. Setup checklist
    a. LOOKUP CONDITION ON STUDY DESIGN PLAN and WRITE ON PAGE 2
    b. Have sent pre-survey
    c. Put water bottle and pencil on table.
    d. Put piles of papers on chair away from table, next to you. Papers to have
        i. On table:
            1. Pen
            2. Water bottle
            3. Phone (to record audio)
                a. Check battery is sufficient
        ii. Aside, next to you:
            1. Warm-up questions (e.g. "Name 10 places people live). 3 sheets of paper
            2. Set of questions
            3. <IF STRATEGY>: face down: strategy sheet, memory table (x3)
            4. Laptop (for post-survey)
        iii. On clipboard:
            1. Protocol
            2. Test 1 under protocol
            3. Answer sheet
            4. Set of questions
        iv. Watch (to keep time)
    e. Have clipboard with problems to take notes on.
    f. Have phone/recording device ready
    g. Have laptop with sufficient battery, with initial study short survey open, set aside.
2. Logistics
    a. "Hi <name>, thanks for coming! Here's a water for you. How are you feeling today?"
    b. Were you able to complete the pre-survey?"
        i. If no: "Let's have you take the survey now."
            1. If asked, it is for javascript
        ii. SURVEY START TIME: _____
        iii. SURVEY END TIME: _____
    c. Just as a reminder, all the information we will gather in the study is for research purposes only. No personally identifiable information will ever be shared outside of the research team. Do you have any questions about that?
    d. Great, we have an hour and a half for the study. If for some reason we need to take more time, how much longer can you stay?
    e. Okay, thank you. Let's start by having do this quick survey.

f. Now, for a few logistical details:
    i. “Would you be willing to turn off your phone during the study?”
g. I’ll ask you some questions today and I want to focus on listening to you. It's really hard for me to take notes quickly enough. Do you mind if I record us talking?
    i. Start recording

3. Study Intro
    a. The objective of this study is to understand your thought process as you complete an assessment of your program tracing skills. We are trying to develop a high-quality assessment of program tracing skills to help improve learning.
    b. This will involve you taking an assessment now, then I will use that to make a learning lesson personalized to you. After that, we will do another assessment.
    c. “Later you will have an opportunity to learn about your results and to learn what you were less strong on.”
    d. "Your performance on the assessment does not impact any of your class grades and personally identifiable information will never be shared with anyone outside of the research team."
    e. "Do you have any questions?"
    f. (notes about you can stop at any time if uncomfortable etc.)
    g. LEARNER’s CONDITION: For think-aloud second
        i. START STRATEGY TIME: _____
        ii. We’ll start with taking the assessment. First, you’ll read the first page of the test, then let me know when you are done.
        iii. WAIT
        iv. Next, I’m going to show you a strategy for answering questions. It helps you work carefully and not make small mistakes. First, you’ll read over the strategy, then I will show several examples of how to use the strategy. Then you’ll try the strategy on some examples and you’ll see how I used the strategy on those examples.
        v. Go ahead and read the strategy and the memory table worksheet.
        vi. Now, I’ll show you an example. (show example with just variable setting, point at the strategy list, x=5 y=1 y = 2)
        vii. Now you try it for the same problem.
        viii. Now, we’ll go through 2 more examples, one at a time. First, you’ll try the example. If it involves parts of the programming language you are less strong with or have never seen, just do your best when you try and don’t worry if you don’t finish. After 30 seconds, I’ll show you how I would do the example. Then we’ll go to the next example. Focus on practicing the strategy with the tables. (pause) go ahead and do the first example.
        ix. You only need to use a new table for function calls, nothing else.
        x. Eg 2
        xi. You only need to use a new table for function calls, nothing else.
        xii. END STRATEGY TIME: _____

xiii. PRETEST START TIME: _____
xiv. PRETEST FINISH TIME: _____

AFTER THE PRE-TEST
Participant's Name: _____ Study Date: _____ Current Time: _____

Thank you for taking the assessment. We'll have a break shortly. Please answer this quick survey first.

Great, thank you again for your effort, I know the test can be tiring and you deserve a break :) Feel free to use the bathroom and walk around and relax and come back by <time + 5 mins from now> - if you have to take longer that is okay too. I'll go over your work and make a short lesson to help you with a few parts you were less strong on. After that, you'll take the second assessment. Any questions or concerns?

ALGORITHM FOR PREPARING INTERVENTION

4. Intervention
    a. From menu, this tutor, program 2
    b. Then the content
    c. WRITE THE CONTENT YOU CHOSE DOWN HERE
        i. 1
            1. Start:
        ii. 2
            1. Start:
        iii. 3
            1. Start:
            2. End:

Okay, now we'll do the second assessment. Please take this quick survey first.

GO TO NEXT PAGE FOR THINK ALOUD

5. Think aloud
    a. THINK ALOUD START TIME: _____
    b. This part of the session today will consist of a think-aloud where you will work through the assessment and verbalize your thinking. Although I will not be able to help you directly here, I will be able to understand your thought process."
    c. "Now I'm going to explain what think-aloud is, and you'll get some practice."
    d. "For the following tasks, I'm interested in what you think about when you find answers to some questions that I am going to ask you to answer. In order to do this, I am going to ask you to THINK ALOUD as you work on the problem given."
    e. "What I mean by think aloud is that I want you "to tell me EVERYTHING you are thinking from the time you first see the question until you give an answer. I would like you to talk aloud CONSTANTLY from the time I present each problem until you have given your final answer to the question."
    f. "I don't want you to try to plan out what you're going to say or try to explain to me what you are saying. Just act as if you are alone in the room speaking to yourself. To help with this, I'll stand behind you after you start doing problems."
    g. It is most important that you keep talking. If you are silent for any long period of time I will ask you to talk by saying "Please Keep Talking". Do you understand what I am asking of you? <WAIT>
    h. Good, now we will begin with some practice problems. First, I want you to answer a question and tell me what you are thinking as you develop an answer. I will NOT be able to answer any of your questions. Just do the best you can. So, answer that problem and tell me what you are thinking as you get an answer. <GIVE PRACTICE SHEET, stand behind. Note time in margin of this page.>
    <when done step back into view (forward) and talk from the right side>
    i. Good. Now, I will give you more practice problems before we proceed to the assessment. I want you to do the same thing for each of these problems. I want you to think aloud as you think about the question (just like you have been doing). After you are done with a question, go on to the next one.
    Any questions? Go ahead and start the next question.
    <TAKE PREVIOUS SHEET. GIVE SHEET, stand behind>
    j. <INTERRUPT WITH FEEDBACK IF NARRATING OR EXPLAINING, OR TOO QUIET>
    <Stop after 2 problems if participant understands think aloud. Continue to 3rd if not>
    k. Good. As a reminder, You don't have to talk to me or explain what you are doing to me. Just say out loud the thoughts that come into your mind as if you are alone by yourself."
    l. WHEN AT THE STRATEGY: Now you will watch a video to learn the tracing strategy and see an example"Good, now, continue thinking aloud as you do the assessment."\

THINK ALOUD END TIME: _____

Now, you'll think aloud, starting from when I hand you the assessment. Once you have the assessment, go through and do the assessment.
ASSESSMENT 2 START TIME: _____
ASSESSMENT 2 FINISHTIME: _____

Post-survey

m. "Great and thank you again for your participation. Please fill out a post-survey." Please sign in with your UW email address. While you do that, I'll look over your assessment to review it with you."
n. WAIT, review answers out of sight
o. Great. Now let's see what you're less strong on and review it with you.
p. (see if they don't know something / confirm guesses or slips)

AFTER THE POST-TEST

Participant's Name: _____ Study Date: _____ Current Time: _____

6. Interview & follow-up questions
    a. "This concludes the assessments and think-aloud. Thank you for your participation! Now I'm going to ask you a few follow questions."
    b. <READ QUESTIONS OFF OF POST-STUDY QUESTIONS>

POST-Study Questions

(Researcher: Feel free to ask reasonable follow-up questions. Ask about anything you're uncertain about)

TIME: _____

7. How did the tutorial in-between the assessments affect your knowledge?

8. What, if anything, surprised you during the tutorial?

9. What, if anything, did you learn from the tutorial?

Write down any follow-up questions and answers:

TIME: _____

TIME: _____

How did taking the test affect you?

How did the tracing strategy with the tables influence your performance?

Did the tracing strategy make it harder to do the problems or confuse you?

Besides what you may have learned in the tutorial, did you do anything differently for the 2nd assessment compared to the 1st assessment?

How did the difficulty level of the first assessment compare to the second? (follow-up: Why?)

(Write follow-up questions below)

TIME: _____

Do you think you will use the strategy with the tables again?

(If yes, follow up:) When would you use this strategy?

(If not:) What strategy would you use instead?

What do you think are the **advantages** of this strategy?

What do you think are the **drawbacks** of this strategy?

TIME: _____

Do you have any anxiety or other concerns after taking the test?

Great, thank you again for your participation. Do you have any questions for me?

TIME: _____