ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/288002756

Do we know how difficult the Rainfall Problem is?

Conference Paper · November 2015

DOI: 10.1145/2828959.2828963

CITATIONS
11

READS
248

5 authors, including:

Otto Seppälä
Aalto University
19 PUBLICATIONS **774** CITATIONS

SEE PROFILE

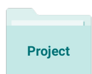
Petri Ihantola
Tampere University of Technology
54 PUBLICATIONS **565** CITATIONS

SEE PROFILE

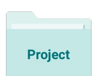
Juha Sorva
Aalto University
32 PUBLICATIONS **444** CITATIONS

SEE PROFILE

Some of the authors of this publication are also working on these related projects:



Gaze in Programming View project



Infrastructure for Computer Science Education View project

All content following this page was uploaded by Juha Sorva on 04 March 2016.

The user has requested enhancement of the downloaded file.

Do We Know How Difficult the Rainfall Problem is?

Otto Seppälä
Aalto University
Espoo, Finland
otto.seppala@aalto.fi

Petri Ihantola
Tampere University of Technology
Tampere, Finland
petri.ihantola@tut.fi

Essi Isohanni
Tampere University of Technology
Tampere, Finland
essi.isohanni@tut.fi

Juha Sorva
Aalto University
Espoo, Finland
juha.sorva@aalto.fi

Arto Vihavainen
University of Helsinki
Helsinki, Finland
avihavai@cs.helsinki.fi

ABSTRACT

The programming task known as the Rainfall Problem has developed a reputation for being surprisingly difficult for introductory-level (CS1) students. We contribute a survey of studies of the problem as well as a new study of students' solutions collected at three institutions. In all three CS1s, at least about half of the students were able to fully solve the problem and the large majority were at least close. Failure to handle invalid or missing input accounted for most bugs. Our survey and study together suggest that the Rainfall Problem is not necessarily overwhelmingly difficult: Success rates vary and some reasonably good results have been achieved under multiple programming paradigms. We provide a breakdown of confounding factors and suggest improvements and hypotheses for future studies of the Rainfall Problem.

CCS Concepts

•**Social and professional topics** → **CS1; Student assessment;**

Keywords

Rainfall Problem, novice programmers, CS1, benchmark

1. INTRODUCTION

The *Rainfall Problem* is a programming task that has been used in a number of studies of programming ability over the past few decades. Here is one variant of the problem: "*Write a program that processes an input consisting of daily rainfall measurements (non-negative integers) until it encounters the integer 99999. The program should output the average of the numbers encountered before 99999.*"

The Rainfall Problem is straightforward enough that many educators have expected their students to be able to solve it at the end of an introductory programming course (CS1).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Koli Calling 2015, November 19 - 22, 2015, Koli, Finland

© 2015 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ISBN 978-1-4503-4020-5/15/11…$15.00

DOI: http://dx.doi.org/10.1145/2828959.2828963

Contrary to such expectations, however, some studies have reported unsatisfactory student performance, a fact that is sometimes used to illustrate the scale of the challenge faced by students and teachers of programming. In 2011, Guzdial commented on the literature on the Rainfall Problem, much of which dates from the 1980s:

> "In one study, only 14% of students in Yale's CS1 could solve this problem correctly. The Rainfall Problem has been used under test conditions and as a take-home programming assignment, and is typically graded so that syntax errors don't count, though adding a negative value or 99999 into the total is an automatic zero. Every study that I've seen (the latest in 2009) that has used the Rainfall Problem has found similar dismal performance, on a problem that seems amazingly simple." [7]

This sentiment has been echoed by other researchers (e.g. [19, 27]):

> "The general conclusion is that large numbers of students are still making the same sorts of error that they were making 30 years ago." [19]

> "Students struggle with this problem today as they did in 1982." [27]

Two recent studies of novice programmers' performance on the problem [19, 6] have reported contrasting results, leading researchers to hypothesize on the impact of GUIs in CS1, design-driven pedagogy, functional programming, and other factors. Failing to locate a survey of success rates that would help us assess claims about Rainfall, we conducted our own, which we report on here. We wished, furthermore, to examine CS1 students' Rainfall performance at our home institutions and thereby to add to the existing pool of empirical studies.

The first two of our research questions, below, pertain to the literature review and the other two aim to partially replicate earlier studies at different institutions:

RQ1 *What success rates have been reported on the Rainfall Problem?*
RQ2 *Which factors vary between the Rainfall studies?*
RQ3 *To what extent do CS1 students at our universities succeed in solving the Rainfall Problem?*
RQ4 *Which subgoals of the problem prove the most difficult for students at our universities?*

The next section outlines the theoretical framework which inspired the original research on the Rainfall Problem and also impacts on the present work. Section 3 presents the results of a literature survey that addresses our first research question. In Section 4, we outline our empirical study, whose results we then present in Section 5. In Section 6, we discuss our findings. Section 7 concludes the article.

2. SCHEMAS, GOALS, AND PLANS

In cognitive psychology, a *schema* is a mental structure that contains generic conceptual knowledge [14]. Solution patterns can be stored in long-term memory as schemas, which people draw on as they encounter new situations and solve problems. Such problem-solving schemas provide "canned" solutions which may be applied to new instances of a familiar class of problems. The formation of problem-solving schemas is a crucial aspect of the growth of expertise: Experts have numerous schemas for a wide variety of problems which they can retrieve from long-term memory and combine to solve complex problems without experiencing cognitive overload.

A number of studies support the idea that programmers make use of schemas as they read and write programs (e.g., [15, 23, 21, 18, 3, 1]). For instance, Rist's [18] studies suggest that programmers use top-down, forward-developing (i.e., linear writing), breadth-first strategies for writing programs whenever possible, that is, whenever they can apply an existing schema to a goal or subgoal. Novices have few schemas and are often forced to rely on bottom-up strategies.

In the 1980s, Elliot Soloway and his colleagues approached computing education research from a perspective inspired by schema theory. They analyzed programs and programmer behavior in terms of the *goals* and subgoals that the programmer was trying to achieve, and the *plans* and subplans that provide the corresponding solutions [20, 26, 25, 2]. The Rainfall Problem can be analyzed in terms of the following subgoals, for which we use, throughout this article, the same names as Fisler [6]:

SENTINEL Process input until the sentinel.
NEGATIVE Ignore negative inputs.
SUM Total the inputs.
COUNT Determine the number of the inputs.
DIVZERO Handle cases with zero inputs.
AVERAGE Compute the average.

Any of a number of subplans can be used to address a subgoal. For example, the subgoal SUM may be addressed by incrementing the value of a variable within a loop or by using the sum method of a list. Subplans are "glued together" in different ways, such as abutment, nesting, and merging [20].

This article relates to Soloway's work on programming plans in three ways. First, the particular task that we investigate, the Rainfall Problem, originates from Soloway's studies of plans. Second, we use the concepts of goal and plan as tools for the post-hoc analysis of students' programs. And third, Soloway's group provided the first empirical results that showed how students struggled with Rainfall. We discuss these findings next.

3. SURVEY: NOVICES AND RAINFALL

In this section, we review earlier studies of the Rainfall Problem. We limit our survey to studies in which the participants were novice programmers (typically first-year university students) and which report success rates on the problem. In several of the studies reviewed, the success rates were incidental to, rather than its main focus of, the study.

Let us begin by discussing variations of the problem. We will then describe each study that falls within the scope of our review before summarizing the answers to our first two research questions.

3.1 Variations of the Rainfall Problem

Different names have been used for the Rainfall Problem in the literature, including "Rainfall Problem," "Averaging Problem," and "Noah Problem." Moreover, there is no standard form of the problem that these names refer to; instead, there are many variations with somewhat different requirements. Below, we describe the main variations. Table 1 provides a detailed mapping of the studies to the variants.

The most well-known version of the problem asks the student to filter out negative inputs. Some variants do not have this requirement. In yet other variants, negative numbers count as zeros or sentinels. Some of the problem statements are vague about how to handle invalid inputs.

The original Rainfall Problem is based on console I/O and requires a full program to be written, but some recent variants instead request a function that receives a collection of input data as a parameter.

Most variants require SENTINEL, but a version in which this subgoal was optional has also been used.

Some variants include additional statistics to be produced besides the average, such as maximum rainfall or the number of rainy days.

Non-functional aspects of the problem statements also vary significantly, as do external constraints. For instance, are example runs or tests provided and for which cases? Are students required to write their own tests? Is the problem answered on paper or on the computer? Does the student have access to a book or the Internet? Can they ask for help from others? Is there a time limit?

3.2 Rainfall Studies

This subsection briefly describes each of the studies included in our review. Selected aspects of these studies are more systematically summarized in Table 1.

3.2.1 Early Studies: 1980–2000

The Rainfall Problem was introduced in the early 1980s as one of three "simple looping problems" typical of introductory programming courses [24]. Soloway and his colleagues used it in a non-credit quiz at the end of CS1 and found that of 31 participants, only 23 produced an answer and that only nine of those answers were correct [24]. The performance of students taking a second programming course (CS2) was only marginally better.

The authors hypothesized that looping strategies play a role in the outcomes and subsequently evaluated this claim by posing the problem to two groups of CS1 students [22]. In a group using regular Pascal, eight of 58 CS1 students (14%) solved the problem. In another group that used a custom variant of Pascal designed to make looping more intuitive, 14 of 58 students (24%) succeeded. The experiment was also carried out in CS2, where the students performed better, reaching success rates of 36% and 61%, respectively. Although the treatment group had somewhat more detailed materials available than the control group, these results support the hypothesis that the enhanced version of Pascal

was beneficial.

Johnson et al. [13, 12] subsequently used a challenging variant of the Rainfall Problem as they evaluated a method for automatically identifying bugs in programs produced by novices. Of the participating CS1 students, 11% solved the problem correctly. In the analysis of Johnson et al., problems with reading or printing values also counted as bugs, as did "spurious" (unnecessary) code.

In 1994, Ebrahimi [4] investigated novice programmers' errors and plan compositions, comparing four groups of students who had been instructed using different programming languages. Ebrahimi, too, assessed the students' solutions to a challenging Rainfall variant in a strict way: Students were penalized not only for bugs in functionality but also for spurious and inefficient code. Ebrahimi reported success rates of 5%, 20%, 20%, and 15% for Pascal, C, Fortran, and Lisp programmers, respectively.

3.2.2 *Recent Studies: 2001–2014*

In 2003, Guzdial et al. [9] reported results from an evaluation that included an easy variant of the Rainfall Problem. Fourteen of 113 students (12%) solved the problem correctly and the average partial score was 46%. The course being studied was a newly launched offering for students who did not major in CS or engineering, and the Rainfall Problem was atypical for this course.

De Raadt [2] used the Rainfall Program in the context of pattern-oriented instruction. One group of students received no explicit instruction on problem-solving strategies (i.e., patterns similar to those required for Rainfall). Another group, studied four years later, did. Only one student out of 42 (2%) in the first group fully solved the problem; eleven (26%) were at least close, with at most one missing subplan. In the second group, the corresponding figures were 31% and 49%. On the basis of these results, De Raadt argued that explicit teaching of problem-solving strategies improves CS1 outcomes.

Venables et al. [29] used a variant of the problem in the context of a 2009 study of code-tracing, explaining, and writing skills. Their results indicate that six of 32 CS1 students (19%) had a fully correct solution and a seventh student was close.

Simon [19] studied the Rainfall Problem in the context of event-driven programming. He asked students to solve a function variant of the problem under exam conditions with very limited time available. None of the 149 students had a fully correct solution, and the average partial score for the question was 23% [19].

Porter et al. [17] investigated whether students' CS1 performance could be predicted from their answers to clicker questions. As a by-product of this investigation, they report on students' performance on a two-part assessment that consisted of the Rainfall Problem and a separate, simpler task. The authors do not provide figures concerning Rainfall alone but they do report that the students' median score was 8 out of 10 points on the combined task.

Fisler [6] studied students' solutions to a function-based variant of the Rainfall Problem using a method similar to that of Ebrahimi [4]. The solutions were collected from multiple institutions and CS1 courses at the university and high school levels. Pooled together, roughly half of the students solved the problem fully correctly, using a variety of strategies for composing subplans. There were significant differences between the contexts studied by Fisler but they all shared a pedagogical approach based on functional programming and systematic program design and testing [5].

3.3 RQ1: Overview of Success Rates

Table 2 shows novices' success rates on the Rainfall Problem as reported in the literature. As the table indicates, many of the students in Fisler's study [6] succeeded in producing a bug-free Rainfall solution. The results reported by Porter et al. [17] also suggest that the problem was not overwhelming. De Raadt's [2] experiment showed a sizable gain in student success as a result of pattern-oriented instruction. The success rates reported in the other studies could be described as worrying, if we make the assumption that the Rainfall Problem has been an appropriate assessment for the students the stage of learning that was investigated by the researchers.

Several of the reported success rates are low, but overall, we would describe the results as mixed. The variation in the figures is difficult to explain because of the different variants of the problem (see above) and other differences between the studies (see below).

3.4 RQ2: Variation in the Rainfall Studies

As Table 1 illustrates, there is an immense amount of variation between the various studies that have reported success rates on Rainfall. Different problem statements aside, the studies also differ in terms of the conditions under which students undertook to solve the problem, the courses that were studied, student demographics, and the way researchers analyzed the solutions, among other things.

Of the studies within the scope of our review, only two experimented on a single variable within the same pedagogical context: The study by Soloway et al. [22] highlighted the usefulness of a particular looping construct for solving Rainfall, and the study by de Raadt [2] supports explicit instruction in problem-solving strategies. In addition, some of Ebrahimi's [4] results may be attributable to different programming languages. We found no comparisons of, say, Rainfall performance in paper vs. computer exams. The number of studies is not sufficient for conducting a quantitative meta-analysis.

Moreover, many of the pedagogical contexts are very loosely outlined in the articles. Although it is not possible report on every variable in detail, there are measures such as CS1 workload that can be quantified but which have not been reported. This makes comparisons more difficult still.

To summarize, we have been able to identify a number of factors that differ between the studies of the Rainfall Problem and that may influence success rates. Nevertheless, we must conclude that the extant literature does not enable us to draw firm conclusions concerning the relative importance of these factors.

4. DESIGN OF THE EMPIRICAL STUDY

In this section, we describe the version of the Rainfall Problem that we used, the three courses that we studied, and the way we collected data and analyzed it.

4.1 Version of the Rainfall Problem

Students were asked to print out the average or report that it could not be computed. They were to accept but

Table 1: Contexts and variation in studies with novices solving the Rainfall problem

		Variant of the Rainfall Problem									Problem Statement			Pedagogical Context							Student Demographics	
Source	Cohort	AVERAGE	NEGATIVE	SENTINEL	DIVZERO	Report rainy day count	Report max/min rainfall	P(rogram)/F(unction)	Input	Notes	Reacting to no input	Example runs / input data	Tests	Level	Week conducted / Weeks in course OR fraction of Course/Semester length	Programming language	Paradigm I(mperative)/F(unctional)	Pedagogical choices	When S(pring)/F(all)/ * Publication. Year	Course content vs. Rainfall	Students' major	Past programming experience
This Article	Context 1	Y	Y	Y	Y	N	N	P	C		print	ANSD	-	CS1	W9/9	Python	I	see the body text	S2013	EP	various Engin.	mostly none
	Context 2	Y	Y	Y	Y	N	N	P	C		print	ANSD	-	CS1	W7/7	Java	I 10)		F2014	P	36% CS	17)
	Context 3	Y	Y	Y	Y	N	N	P	C		print	ANSD	GA	CS1	W5/14	Python	I		F2014	EP	55% CS	mostly none
Fisler [6]	T1	Y	Y	Y+L	Y	N	N	F	L		unspec.	?	T	CS1	W13	Racket, Ocaml	F	HtDP	F2013	?	CS	?
	T1Acc													CS1	W13	Racket, Pyret	F	HtDP	F2013	?	CS	?
	T2													CS1	W10	Racket	F	HtDP	F2013	?	CS	?
	T3Non													CS1	W10	Racket	F	HtDP	F2013	?	Non-CS	?
	HS													CS1	W18	Racket	F	HtDP	F2013	?	High School	?
Porter et al. [17]		Y	S	S	Y	N	N	P	C	1, 3)	unspec.	ASD 4)	-	CS1	W12/12	Python	I	11)	2013	?	?	?
Simon [19]		Y	Z	Y	Y	N	N	F	A	2)	unspec.	NS	-	CS1	ES	C#	I	12)	2013*	X	IT	?
Venables et al. [29]		Y	S	S	Y	N	N	P	C		print	ANSD	-	CS1	ES?	Java	I	?	2009*	?	?	?
De Raadt [2]	2007	Y	I	Y	Y	N	N	P	C		unspec.	-	-	CS1	?	C	I	13)	2007	EP	mainly IT, Engin	?
	2003													CS1	6)	C	I	?	2003	?	mainly IT, Engin	?
Guzdial et al. [9]		Y	Y	O Y+L	?	N	N	F	L	3)	unspec.	NS	-	CS1	7)	Python	I	14)	S2003	15)	Non-CS	?
Ebrahimi [4]	Pascal	Y	Y	Y	Y	Y	Y	P	C		unspec.	ANS?	-	CS1	W6/S	Pascal	I	?	1994*	EP?	?	-
	C													post CS1	W4/S	C	I	?		EP?	?	some
	Fortran													post CS1	W4/S	Fortran	I	?		EP?	?	some
	Lisp													post CS1	W4/S	Lisp	I	?		EP?	?	some
Johnson et al. [12, 13]		Y	Y	Y	Y	Y	Y	P	C		unspec.	-	-	CS1	?	Pascal	I	?	1983*	?	?	none
Soloway et al. [22]	Nov.-Pascal	Y	I	Y	N	N	N	P	C		unspec. but not required either	-	-	CS1	¾ S	Pascal	I	?	1983*	16)	?	?
	Nov.-Pascal-L													CS1	¾ S	Pascal-L 9)	I	?	1983*	16)	?	?
	Adv.-Pascal													CS2	⅔ CS2	Pascal	I	?	1983*	?	?	1 course
	Adv.-Pascal-L													CS2	⅔ CS2	Pascal-L 9)	I	?	1983*	?	?	1 course
Soloway et al. [24]	Novices	Y	I	Y	N	N	N	P	C			?	-	CS1	ES 8)	Pascal	I	?	1982*	?	?	none
	Intermediates													post CS1 5)	W10/16	Pascal	I	?	1982*	?	?	1 course

Y The plan was required in its standard form.
N The plan was not required in this variant.
Z Negative values were included as zeros.
I Negative values were included in the total.
S All negative values acted as sentinels.
Y+L Either the sentinel or the end of the list terminates input
O The students could choose whether to implement SENTINEL
C/A/L Input from Console/Array/List
P/F What to implement: Program/Function

1) Using lists, tuples, or dictionaries not allowed.
2) Using a for loop was not allowed.
3) The problem definition was ambiguous about zeros as values.

Examples shown contained:
A the resulting average
N negative values
S a sentinel value
D no non-negative input
Unspec. The student decides.
4) All negative values were sentinels

Tests
GA Students were given automated tests with partial coverage and limited feedback.
T Expected students to submit tests

General
? Not reported / Unknown
- None

Course
5) 2nd course in programming (data str.)

Timing
S Semester **W** Week **ES** End of Semester
6) Third last week
7) 2nd Midterm
8) Final week of a summer session course

Language / Pedagogy
HtDP How to Design Programs
9) Pascal with modified loop constructs
10) Object-oriented
11) Peer Instruction, Pair Programming, Contextualization, Clickers
12) GUI-driven
13) Explicit strategy instruction (including guarded div)
14) Media Computation

Course content vs. Rainfall
E Course contains examples similar to Rainfall.
P Course contains problems similar to Rainfall.
X Rainfall was modified to better match content.

15) List processing had not been covered.
16) had exp. with while and other loop constructs.

Previous programming experience
17) 56% never programmed before
33% programmed little, median ~100h
11% programmed more, median ~200h

ignore negative inputs. Two annotated example outputs were given: 1) a run with three positive inputs, a zero, and a negative number before the sentinel; 2) a run in which the sentinel was the first and only input. Students did not need to use exactly the same output formatting as in the examples, except in Context 3 where automatic assessment was provided for Rainfall.

4.2 Contexts and Data Collection

Solutions to the Rainfall Problem were collected in three CS1 courses at three institutions as described below and summarized in Tables 1 and 2. We were opportunistic: In one context, we happened to already have data available from an unrelated project, and in the other two contexts, we introduced the Rainfall Problem to an ongoing CS1 in a way that was deemed appropriate by the teacher.

4.2.1 *Context 1: Aalto University, CS1 for Engineers*

We studied a large-class CS1 for students who major in various branches of engineering (excluding computing). Most of the students had little to no prior programming experience, and their degrees typically require only this one course. In Spring 2013, the course consisted of nine weeks of lectures (4 hours per week) which ran in parallel with nine rounds of programming assignments.

Students worked on the assignments in open labs at their own pace but with weekly deadlines. Help was available from undergraduate teaching assistants on request. In terms of content and ordering, the course is a very traditional CS1 that starts from variables, operators, and selection, then turns to iteration and functions. Examples that computed the average of inputs, one of which was a simpler variant of the Rainfall Problem, were used during the early weeks (roughly two months before our data collection). All the programming assignments were closed-ended: They required students to write small programs that produced console output as specified. Students submitted their programs to a web-based system for instant feedback on functionality and automatic marking. To pass the course, the students had to gain at least half of the available marks during each of the first eight weeks. The present authors were not involved in teaching this course.

Table 2: Results from publications/data sets listed in Table 1

Source	Cohort	Results: Number of Students participated/submitted	Number of students in course/class/total	Fully correct percentage NEGATIVE required	Fully correct percentage NEGATIVE not required *	Correct when DIVZERO ignored	Nearly correct 7) percentage	No correct subplans / No (pertinent) code	Average partial score	Experimental Setting: Setting	(T)ake-home / (L)ab	Material / Help Available	Time limit (mins)/(E)xam	Paper / Computer	C(ompiler) / I(DE)	Incentive	Notes
This Article	Context 1	151	> 550	45 %	n/a	66 %	+37%	0 %	88 %	S	L	N	60	C	I	PR	
	Context 2	192	243	53 %		70 %	+36%	1 %	89 %	A	T	M	U	C	I	P	
	Context 3	165	236	72 %		77 %	+20%	0 %	94 %	A	T	H,Pair	U	C	I	P	
Fisler [6]	T1	61	154	54% 1)	n/a	74%1)	+21-23%4)	3-6%4)	?	A	T	M	U	C?	?	G?	
	T1Acc	44	44	39% 1)		52%1)				A	T	M	U	C?	?	G?	
	T2	63	224	11% 1)		22%1)				E	n/a	N	E	P?	n/a	G?	
	T3Non	43	65	2% 1)		14%1)				A	L	H	10	C?	?	G?	
	HS	7	7	0% 1)		5%1)				A	L	H	20	C?	?	G?	
Porter et al. [17]		>100 2)	>100 2)	n/a	?	?	?	?	6)	E	n/a	N	E	P?	n/a	G	
Simon [19]		149	?	0 %	n/a	?	?	36 %	23 %	E	n/a	N	9)	P	n/a	G	
Venables et al. [29]		32	?	n/a	19 %	?	+3%	18%	48 %	E	n/a	N	E	P?	?	G	
De Raadt [2]	2007	45	?	n/a	31 %	49 %	?	?	69 %	S	n/a	N	U	P	n/a	N	
	2003	42			2 %	24%		2%	57 %							N?	
Guzdial et al. [9]		113	120	12%**	n/a	?	?	?	46%**	E	n/a	N	E	P?	n/a	G	
Ebrahimi [4]	Pascal	20	?	5%	n/a	?	+5%	5)	?	S	n/a	?	?	C	C	?	
	C	20		20%			+0%										
	Fortran	20		20%			+0%										
	Lisp	20		15%			+10%										
Johnson et al. [12, 13]		206	206	11 %	n/a	?	?	?	?	?	?	?	?	C	C	?	10)
Soloway et al. [22]	CS1-Pascal	58	?	n/a	14 %	?	?	?	?	S	n/a	M8)	U	P	n/a ?	?	
	CS1-Pascal-L	58			24 %												
	CS2-Pascal	53			36 %												
	CS2-Pascal-L	59			61 %												
Soloway et al. [24]	Novices	23	31	n/a	39 %	39% 3)	?	?	?	Q	n/a	N	?	P	n/a	N	
	Intermediates	43	52	n/a	42 %	42% 3)											

Results

1) Information received via personal communication with author
2) The number of students was not reported but scatter plots have at least 100 data points
3) DIVZERO was not required
4) Only programs with a clear plan structure included (187 submissions)
5) For exact error counts per language construct see [4]

* Negative values were not filtered (interpreted as sentinels, zeroes, or valid values)
** Without sentinel

6) Total median 8 out of 10 for *two* assignments (one of which was Rainfall)
7) in addition to (either of) the fully correct precentages

Material/help available

M Materials available
n/a Not applicable
Pair Pair programming allowed
H Help available
8) Material demonstrated loop constructs
N Nothing

Setting

E Exam
A Assignment
S Study
Q Quiz

Reward

P points (fixed)
G points (graded)
R reward (e.g. money)

Time limit

U Unlimited time
E Within exam

9) Exam 3 hrs, some 20 min for this task in particular [personal communication]
10) First syntactically correct

Data collection: Rainfall data was collected near the end of the course, as part of another, broader research effort. Volunteers received a small bonus to their course assessment plus a movie ticket. Instructions were given on paper by a research assistant. Students used a university computer to solve the problem, which they had to do without recourse to a textbook, the Internet, automated tests, or other resources. The problem statement contained a reminder of the library functions that they could use to read and parse input. A time limit of approximately one hour was given; apart from that, students were free to decide for themselves when they were finished. The reward was given irrespective of success or lack thereof. Of the over 500 students in the course, 151 participated. The participants scored slightly lower on the CS1 examination than the non-participants, but the difference was not statistically significant (non-normal distribution; Wilcoxon rank-sum test, $p \approx 0.20$).

4.2.2 Context 2: University of Helsinki, Generic CS1

We studied a seven-week CS1 in Java, open to all students of the university. In Fall 2014, about 40% of the students were first-year CS majors, for whom the course is mandatory; the others had elected to take it. Most of the students had no prior programming background.

The first three weeks focused on imperative programming: Students practiced writing dozens of small programs using variables, iteration, methods, lists, and console I/O. Later, objects were introduced and students practiced writing object-oriented programs to an input-output specification. Several assignments featured elements of the Rainfall Problem, such as averages. The course had a weekly 2-hour lecture, but most student work was done either at home or in class under the guidance of undergraduate TAs. Most programming assignments were closed-ended, and over 90% of them were computer-assessed, with instant automatic feedback avail-

able. One of the authors of this article was the teacher of the course.

Data collection: The Rainfall Problem was the first programming assignment of the seventh week of the course, and was handed out with the other programming assignments for that week. Contrary to most of the course assignments, the problem was not automatically assessed, and the problem statement indicated that the student could submit their solution regardless of whether they created a fully working solution or not. Students received a minor amount of course credit for submitting any attempted solution. The students could attempt the problem at home or in class, and had access to the Internet and course materials, but the teaching assistants were instructed not to help students with this problem. No time limit was imposed. Out of the 243 students who started the course, 201 were active during the final week and 192 of them submitted a solution to Rainfall.

4.2.3 Context 3: Tampere University of Technology, Generic CS1

We studied a Python CS1 that is mandatory for almost all students of the university, regardless of major. The majority of the course participants had no prior programming background. In Fall 2014, 236 students started the course, about 55% of whom were CS majors. 165 students submitted Rainfall. The course lasted 14 weeks.

The course used a flipped classroom pedagogy and an imperative approach to programming. Averaging programs were used as examples within the supplementary course materials, which were partially the same as in Context 1. Students worked mostly at home or in class under the guidance of undergraduate TAs. They needed to solve multiple small programming assignments every week, all of which were automatically assessed with instant feedback. One of the authors of this article was the teacher of the course.

Data collection: The Rainfall Problem was the first weekly programming assignment of the fifth week of the course. At this point, the course had covered I/O, variables, loop structures and functions; lists had not yet been covered. The students worked on the problem when and how they wished, much as in Context 2, and could use any resources they wished. Working in pairs was allowed but fairly uncommon. In this context, automatic feedback was available, and students could submit a solution as many times as they wanted in order to receive a report from a suite of tests. Not all corner cases were tested by the test suite, but the students were not informed of the deficits in test coverage. We analyzed each student's final submission.

4.3 Analysis

To answer our third and fourth research questions (overall success rate, relative difficulty of subgoals), we needed to locate the bugs in the student programs and determine which subplan each bugs was associated with. We analyzed each solution in terms of the six subgoals of the Rainfall Problem that we listed in Section 2 and which were also used by Fisler [6]: SENTINEL, NEGATIVE, SUM, COUNT, DIVZERO, and AVERAGE. We focused on these six subgoals and ignored the I/O aspects of the programs (which, anecdotally, had extremely few bugs); these subgoals are shared by both the console I/O and function variants of the Rainfall Problem.

In each student program, we searched for the subplans that corresponded to the six subgoals, and classified them as *Correct* (i.e., fully functional), *Incorrect* (i.e., an attempt to address the subgoal, but with one or more bugs) or *Missing* (i.e., no evidence of an attempt to address the subgoal). We did not use a more fine-grained bug classification scheme such as the one used by Fisler [6], with one exception: For the DIVZERO subgoal, we noted which solutions were otherwise correct but had failed to address the corner case in which all inputs are negative and hence invalid.[1] Coding this bug separately enables us to estimate how many of the students would have been able to solve the easier variant of the Rainfall Problem that did not require NEGATIVE.

As noted in Section 3, some earlier researchers have counted "spurious code," i.e., unnecessary additional code, as a bug. We considered such code a flaw in style and focused only on bugs in functionality. Here are some other decisions that we made, largely in line with the earlier literature:

- We ignored the few minor syntax errors that we found.
- An empty submission counted as having all subplans missing.
- We considered it OK to use all the values from 999999 up (rather than just 999999) as sentinels.
- Discarding zeros as invalid was a bug in NEGATIVE.
- Reporting "No input" if all inputs were zero was a bug in DIVZERO.
- If a subgoal was made easier because another subgoal had not been attempted, we gave the student the benefit of the doubt and only penalized them for the missing subgoal. For instance, not attempting to filter out negative values eliminates the possibility of certain bugs in DIVZERO.
- Even if a bug in one plan affected the result of other plans, we only counted the bug once. E.g., if a bug in SENTINEL causes the program to incorrectly ignore some inputs, it also impacts on the end results of SUM, COUNT, and AVERAGE. Still, if those other plans were otherwise fine, we counted them as Correct.
- In the case of Java programs, we coded as Correct the seven solutions that divided a `double` by zero (which evaluates to `Infinity`) before checking the result of COUNT.

The programs were coded by three of the authors while the others cheered them on from the sidelines. To assess inter-rater reliability, we initially chose 10 random programs from each context, 30 in total, all of which were coded by the three researchers. Only a low degree of reliability was reached: the multivariate iota [11] of the categorization of the plans was 0.18 ($p < 0.05$). After discussing points of disagreement, a different set of 30 random programs was selected and analyzed by the same authors. This time, an iota of 0.66 ($p < 0.05$) indicated a good level of agreement. The three researchers then split the remaining programs between them and coded them separately.

An analysis of how students had implemented and combined plans was also carried out and will be reported elsewhere, along with findings from additional contexts. Most solutions in the three contexts discussed here were of the "Single Loop" [6] variety, in which the plans for the SENTINEL, NEGATIVE, SUM, and COUNT subgoals (at least) are merged.

5. EMPIRICAL RESULTS

[1]This is the fairly common bug coded as Missing–Guard–Partial (X-G-P) by Fisler [6].

5.1 RQ3: Overall Performance

Figure 1 summarizes the students' success levels on the Rainfall Problem. As shown in the figure, the proportion of fully correct solutions, that is, programs in which all six subplans worked, was 45.0% in Context 1, 52.6% in Context 2, and 72.1% in Context 3. Looking at "almost correct" solutions, in which all but one subgoal was correctly addressed (i.e., the sums of the two leftmost bars in Figure 1), the percentages rise to 82.1%, 89.1%, and 92.1%, respectively. The average numbers of correct plans for each context were 5.25 ($\sigma = 0.79$), 5.34 ($\sigma = 0.92$), and 5.61 ($\sigma = 0.76$), respectively. A Kruskal-Wallis test indicated no significant differences between Contexts 1 and 2 ($p > .05$) in terms of the number of subplans correctly implemented; Context 3 had more correctly implemented plans ($p < .05$).

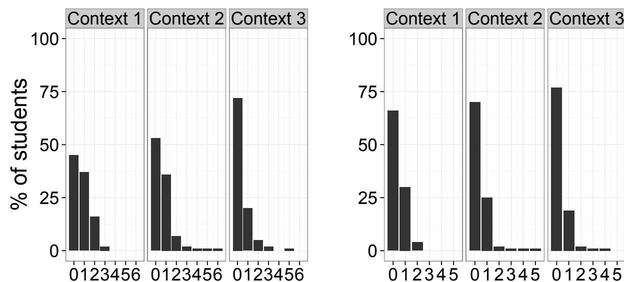


Figure 1: The proportions of Incorrect or Missing subplans. The results for the full problem version are shown on the left and the sans-Negative results (see the body text) on the right.

As noted in Section 3, some variants of the Rainfall Problem do not require students to omit negative numbers. For this reason, we counted the number of fully correct solutions so that we ignored Negative entirely as well as the corner case in which all inputs are negative (which otherwise counted as a bug in DivZero). This produces a lower-bound estimate of how well these students would have been able to solve the easier variant of the problem. As Figure 1 also shows, the proportion of fully correct answers by this metric is in the 66% to 77% range, rising to about 95% across the board if we allow one incorrect or missing subplan.

5.2 RQ4: Individual Subgoals

Table 3 shows statistics for each subgoal.

The DivZero plans had the most issues in every context: 44%, 26%, or 24% of the solutions either did not have a plan for this goal or had a buggy one. The second most difficult plan was Negative.

The remaining subgoals — Sentinel, Count, Sum, and Average — were correctly addressed in the vast majority of student programs.

6. DISCUSSION

6.1 Success Rates in Our Study

The students' overall performance was reasonably good in the three contexts that we studied. The lowest percentage of fully working solutions was in Context 1, which placed strict limitations on outside help and time, and even there roughly half of the students succeeded. Allowing for a single bug, more than 80% of the students succeeded in all three contexts. If we ignore omissions and errors that are obviously related to filtering out negative values (which is not required in all variants of Rainfall), a large majority of all the students solved the problem correctly and nearly all were at least close.

Table 3: The six subgoals listed from most difficult overall to least. The percentages of missing and incorrect plans are shown; M stands for Missing and I for Incorrect.

	Context 1			Context 2			Context 3		
Subgoal	M+I	M	I	M+I	M	I	M+I	M	I
DivZero	44.4	7.9	36.4	25.5	11.5	14.1	23.6	0.0	23.6
Negative	17.2	4.0	13.2	25.5	16.1	9.4	9.1	0.0	9.1
Sentinel	9.3	0.0	9.3	6.8	1.0	5.7	2.4	0.0	2.4
Count	3.3	0.0	3.3	4.7	1.6	3.1	2.4	0.0	2.4
Sum	0.7	0.0	0.7	1.0	0.5	0.5	1.2	0.0	1.2
Average	0.0	0.0	0.0	2.1	2.1	0.0	0.6	0.0	0.6

These figures contrast with the low success rates reported in some of the earlier studies and provide further evidence that acceptable levels of success on Rainfall are being achieved in some contexts. They also illustrate that such results are not limited to a specific programming paradigm or pedagogical approach such as functional programming.

6.2 Possible Factors Behind the Success Rates

There are a great many factors that affect the results. The problem statements we used (which were detailed, with example runs) may have made the problem easier than in some other studies. The student populations will have been different. The fact that some other researchers have counted unnecessary code to be a bug complicates comparisons. Etc. Although we cannot be sure which factors are the most significant, we can put forward some hypotheses that are compatible with our survey and our empirical findings.

The pattern of results was similar, perhaps surprisingly similar, across all our three contexts. While this is likely to be explained in part by similarities between the universities and student backgrounds, it makes sense to also consider the pedagogical commonalities between the three CS1s. These include:

- The students could attend open labs.
- They had numerous weekly programming assignments.
- Those programs used console input (as did the Rainfall variant we used).
- The students' marks depended crucially on completing the assignments.
- The assignments were automatically assessed, with instant feedback on functionality.
- Each of the courses is based on custom materials created by university teachers rather than relying on a textbook.

None of the courses features functional programming and in this respect differ markedly from the contexts studied by Fisler [6], who also reported fairly high success rates on Rainfall. The students had not received explicit instruction on problem-solving patterns, as the higher-achieving students in de Raadt's [2] study had. However, the students in our study did have the advantage that they had been exposed to very similar albeit simpler examples of summing and

averaging earlier during CS1. This will have helped them to form problem-solving schemas that are suited to Rainfall; moreover, students use examples they have seen as templates for solving problems [16]. In Contexts 2 and 3, the students had access to course materials and may have copied parts of the solution from the materials (or other sources).

The highest success rates that we observed were those in Context 3. This result is likely to have been affected by the automatic assessment and feedback that was available to students in that context (cf. [28]). Pair programming may also be a factor here, although most students in Context 3 appear not to have worked in pairs despite it being allowed.

The amount of programming practice that students gain during CS1 significantly impacts their learning outcomes [10], and we suspect that this is a major factor behind some of the Rainfall success rates. The students we studied, for example, had spent many dozens of hours working on CS1, with most of those hours spent on concrete programming practice. The CS1s in many other Rainfall studies may have had a lesser weekly workload in general and a lesser requirement for programming practice in particular. We suggest that in future studies, a focused effort is made to estimate and report the hours that students have actually spent studying and programming in CS1. Where possible, researchers should also quantify students' pre-CS1 practice.

The students that we studied are fairly strong in terms of academic achievement and, perhaps, study skills. This can explain the findings to some extent. Then again, it is unlikely that, say, the original participants from Yale, who had trouble with Rainfall, were weak students either.

6.3 Difficult Subgoals

The same subgoals caused students difficulty in all the three contexts. The subplans that were most commonly missing or incorrect were those related to missing input (DivZero) and the filtering of invalid input (Sentinel). One of the more common bugs involved a combination of the two: the program crashed on all-negative input. Other typical bugs included reporting "No input" when all inputs were zero, and filtering out zeros. Bugs in other subgoals were rare. In broad terms, then, the students usually succeeded in producing a program for averaging inputs, but fairly often failed to deal with one or more special or corner cases.

The students in all three contexts had practiced or at least seen sums, averages, and sentinel-controlled loops, but filtering out invalid inputs (as in Negative) was not a familiar pattern from CS1. This will have impacted on the results to some degree.

The fact that Context 3 had no Missing subplans at all was presumably due to the immediate automatic feedback that informed students of missing parts of their solutions. Context 2 had more missing plans but fewer incorrect plans than the other contexts, a result that we are not able to fully explain. The disproportionately high number of Negative subplans that were Missing in Context 2 may partially explain the relative infrequency of DivZero-related bugs in that context (as not attempting Negative makes DivZero somewhat easier).

The DivZero and Negative subgoals have also been challenging for students elsewhere who worked on other variants of the Rainfall Problem. This is illustrated in Table 4.

Table 4: Aspects of Rainfall solutions in our contexts (C), Fisler's study [6], and Simon's study [19]. (Adapted and extended from [6].)

	Simon	Fisler	C1	C2	C3
Sentinel OK	22%	81–90%	91%	93%	98%
Sum OK	21%	85–89%	97%	95%	98%
Negative OK	40%	57%	83%	75%	91%
DivZero OK	0%	55-61%	56%	75%	86%

6.4 Problem Statements and Testing Skills

Our results, together with those of Fisler [6], show that even students who are largely able to solve Rainfall frequently struggle with special cases. That special cases account for many bugs is of course entirely unsurprising, but it is useful to be reminded of this fact in the context of the Rainfall Problem. The problem does not only require the student to write code but also to form an understanding of the problem in the first place, to pay attention to specifications, to assess which special cases they are expected to cover, and to evaluate their eventual solution.

The danger of students' alternative interpretations of tasks has been noted by other Rainfall researchers: Spohrer [25], for instance, reported that two common alternative interpretations were to leave out the loop or to misinterpret the type of validation required in DivZero and/or Negative. In our study, despite the fairly high student performance overall, input validation bugs were common, and we were left with the impression that some of the participants who had bugs in DivZero or Negative would have been able to implement those plans correctly if they had realized what the intended goal was.

Some students will have read the problem statement carelessly. Some may have simply assumed that it did not matter how, say, all-negative input sets were handled (not explicitly covered by the problem statement). It also seems likely that many of the bugs were left in the students' submissions because of the students' poor testing skills or disinclination to test their programs carefully. Future studies could try to draw these aspects apart. How much do students test their programs? Do students fully understand the problem statement? If we first make sure they do, how hard is the Rainfall Problem then? Do students have more difficulty in realizing the existence of a corner case than in fixing it? Does the use of automated assessment in a CS1 have an effect on these difficulties? We second Fisler's [6] suggestion of studying the test cases that students produce, which could shed light on some of these issues; we additionally suggest logging students' testing behavior within an IDE.

A related issue is whether researchers have unintentionally made Rainfall harder by using an ambiguous problem statement and expecting students to interpret it in a particular way.[2] Especially in an artificial, context-free laboratory assignment such as Rainfall, it may be unrealistic to expect CS1 students to spontaneously realize (or even agree about) the need to cover special and corner cases beyond those explicitly demanded, unless their CS1 teaches and routinely expects such practices. In fact, there is evidence that *experts* who are given the Rainfall Problem sometimes implement only those

[2]Some of the studies have explored students' strategies or bugs rather than success rates. An unambiguous problem statement may not have been necessary or even desirable for all research purposes.

input validation features which are detailed in the problem statement [2].

6.5 A Remark on Code Quality

Although we have established that, for the most part, the students in this study produced programs that work, many of the working solutions, too, left a lot to be desired. Formatting issues aside, the programs commonly had poor organization and the subplans were haphazardly glued together. Many programs gave us the impression that if there had been just one more requirement and the student had had to merge in one more subplan, they would have been likely to struggle and produce a bug. This observation, together with the results from the study, suggests to us that these courses, and other similar contexts elsewhere, would benefit from a concentrated effort to improve students' skills in testing, specification, plan composition, functional abstraction, and coding style.

6.6 Survey of Rainfall Results

Our review of the Rainfall Problem in Section 3 demonstrates that the studies differ from each other in terms of a vast number of potentially significant factors. These factors relate variously to research design and implementation, the participants, the differences in the goals and pedagogies of introductory computing courses, as well as the use of different variants of the problem.

Although some of the success rates on Rainfall have been low, this is not the case in all studies, including our own. This, in combination with the small number of studies and the great differences between them, means that no big picture emerges from the literature concerning the difficulty of the Rainfall Problem for CS1 students. This is, unfortunately, the main overall finding from our literature survey.

Given this result, we believe that it is imperative to conduct further research on the Rainfall Problem if the problem is to be used as a credible benchmark. At least until such research is carried out, statements about the difficulty of the problem across many present-day CS1s require qualification.

To reach a more nuanced understanding, future studies need to report on teaching contexts and student demographics in detail. Our survey in this article contributes a systematic breakdown of potentially relevant variables for consideration; we suggest that researchers who look into Rainfall report on these factors at least. Studies could also be designed to target particular variables.

6.7 Limitations of Our Study

Regrettably, we have not been fully able to follow our own advice concerning the reporting of contextual information. This is because of our on-the-fly data collection: We have limited background information about the students and can only roughly estimate the time they spent practicing programming during CS1 before they tackled Rainfall.

Again because of the opportunistic way we collected data, we have not been able to isolate individual factors as one might in an experiment. Our three contexts are dissimilar in a number of ways. Direct comparisons between them remain rather speculative, as do comparisons between our findings and those reported in the earlier literature.

Despite these significant limitations, we believe our findings are valuable to the research community as they provide further evidence of the wide variability in students' Rainfall performance and suggest several hypotheses. If similar research is carried out in other contexts, meta-studies may be conducted in order to draw firmer conclusions.

7. CONCLUSION

The Rainfall Problem is just one programming problem. In and out of itself, it is unimportant. However, the problem has proven popular among computing education researchers and is sometimes cited as a sort of informal benchmark for beginners' programming ability, usually in order to emphasize how students fail to solve the problem.

Our review of Rainfall studies, to which we have added a new study, illustrates the great variability in both studies and success rates across courses and institutions. In addition to demonstrating that reasonably good success rates on Rainfall can be achieved in imperative CS1s, we have drawn attention to various confounding factors in Rainfall studies, such as the amount of time students have spent on programming practice. Moreover, we have highlighted that there are ambiguities in Rainfall problem statements, especially with respect to corner cases, which may in effect mean that students are being assessed on their ability to guess the researchers' or teachers' precise intentions.

These factors should be addressed in future studies and eventually disentangled if the Rainfall Problem is to serve as a useful indicator of programming ability. In any case, complementary problems and metrics are also required.

8. ACKNOWLEDGMENTS

We thank Kerttu Pollari-Malmi for letting us study her students. We thank Teemu Sirkiä and Timi Seppälä for their help with data collection.

9. REFERENCES

[1] S. P. Davies. The role of notation and knowledge representation in the determination of programming strategy: A framework for integrating models of programming behavior. *Cognitive Science*, 15(4):547–572, 1991.

[2] M. de Raadt. *Teaching Programming Strategies Explicitly to Novice Programmers: Can the Way We Teach Strategies Improve Novice Outcomes?* Verlag Dr. Müller, Saarbrücken, Germany, 2009.

[3] F. Détienne. Expert programming knowledge: A schema-based approach. In J.-M. Hoc, T. R. G. Green, R. Samurçay, and D. J. Gilmore, editors, *Psychology of Programming*, chapter 3.1, pages 205–222. Academic Press, 1990.

[4] A. Ebrahimi. Novice programmer errors: Language constructs and plan composition. *International Journal of Human-Computer Studies*, 41(4):457–480, 1994.

[5] M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi. The structure and interpretation of the computer science curriculum. *Journal of Functional Programming*, 14(4):365–378, 2004.

[6] K. Fisler. The recurring Rainfall Problem. In *Proceedings of the Tenth Annual Conference on International Computing Education Research*, ICER '14, pages 35–42. ACM, 2014.

[7] M. Guzdial. From science to engineering — Exploring the dual nature of computing education research. *Communications of the ACM*, 54(2):37–39, 2011.

[8] M. Guzdial. Exploring hypotheses about media computation. In *Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research*, ICER '13, pages 19–26. ACM, 2013.

[9] M. Guzdial, R. Fithian, A. Forte, and L. Rich. Report on pilot offering of CS1315 Introduction to Media Computation with comparison to CS1321 and COE1361. Georgia Tech report cited in [8], 2003.

[10] L. J. Höök and A. Eckerdal. On the bimodality in an introductory programming course: An analysis of student performance factors. In *2015 International Conference on Learning and Teaching in Computing and Engineering*, LaTiCE '15, pages 79–86, 2015.

[11] H. Jansson and U. Olsson. A measure of agreement for interval or nominal multivariate observations. *Educational and Psychological Measurement*, 61(2):277–289, 2001.

[12] W. L. Johnson and E. Soloway. PROUST: An automatic debugger for Pascal programs. *BYTE*, 10(4):179–190, 1985.

[13] W. L. Johnson, E. Soloway, B. Cutler, and S. Draper. Bug Catalogue: I. Technical report, Yale University, YaleU/CSD/RR #286, 1983.

[14] S. Kalyuga. Schema acquisition and sources of cognitive load. In J. L. Plass, R. Moreno, and R. Brünken, editors, *Cognitive Load Theory*, pages 48–64. Cambridge University Press, 2010.

[15] K. B. McKeithen, J. S. Reitman, H. H. Rueter, and S. C. Hirtle. Knowledge organization and skill differences in computer programmers. *Cognitive Psychology*, 13:307–325, 1981.

[16] P. L. Pirolli. Effects of examples and their explanations in a lesson on recursion: A production system analysis. *Cognition and Instruction*, 8(3):207–259, 1991.

[17] L. Porter, D. Zingaro, and R. Lister. Predicting student success using fine grain clicker data. In *Proceedings of the Tenth Annual Conference on International Computing Education Research*, ICER '14, pages 51–58. ACM, 2014.

[18] R. S. Rist. Schema creation in programming. *Cognitive Science*, 13:389–414, 1989.

[19] Simon. Soloway's Rainfall Problem has become harder. In *Learning and Teaching in Computing and Engineering*, LaTiCE '13, pages 130–135, 2013.

[20] E. Soloway. Learning to program = Learning to construct mechanisms and explanations. *Communications of the ACM*, 29(9):850–858, 1986.

[21] E. Soloway, B. Adelson, and K. Ehrlich. Knowledge and processes in the comprehension of computer programs. In M. T. H. Chi, R. Glaser, and M. J. Farr, editors, *The Nature of Expertise*, pages 129–152. Lawrence Erlbaum, 1988.

[22] E. Soloway, J. G. Bonar, and K. Ehrlich. Cognitive strategies and looping constructs: An empirical study. *Communications of the ACM*, 26(11):853–860, 1983.

[23] E. Soloway and K. Ehrlich. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, 10(5):595–609, 1984.

[24] E. Soloway, K. Ehrlich, J. G. Bonar, and J. Greenspan. What do novices know about programming? In A. Badre and B. Shneiderman, editors, *Directions in Human–Computer Interactions*, volume 6, pages 27–54. Ablex Publishing, 1982.

[25] J. C. Spohrer. *MARCEL: Simulating the Novice Programmer*. Intellect Books, 1992.

[26] J. C. Spohrer, E. Soloway, and E. Pope. A goal/plan analysis of buggy Pascal programs. *Human–Computer Interaction*, 1(2):163–207, 1985.

[27] C. Taylor, D. Zingaro, L. Porter, K. Webb, C. Lee, and M. Clancy. Computer science concept inventories: Past and future. *Computer Science Education*, 24(4):253–276, 2014.

[28] I. Utting, D. J. Bouvier, M. E. Caspersen, A. Elliott Tew, R. Frye, Y. Ben-David Kolikant, M. McCracken, J. Paterson, J. Sorva, L. Thomas, and T. Wilusz. A fresh look at novice programmers' performance and their teachers' expectations. In *Proceedings of the 2013 ITiCSE working group reports*, ITiCSE -WGR '13, pages 15–32. ACM, 2013.

[29] A. Venables, G. Tan, and R. Lister. A closer look at tracing, explaining and code writing skills in the novice programmer. In *Proceedings of the Fifth International Workshop on Computing Education Research*, ICER '09, pages 117–128. ACM, 2009.

View publication stats