Modernizing Plan-Composition Studies

Kathi Fisler
WPI
kfisler@cs.wpi.edu

Shriram Krishnamurthi
Brown University
sk@cs.brown.edu

Janet Siegmund
University of Passau
siegmunj@
fim.uni-passau.de

ABSTRACT

Plan composition is an important but under-studied topic in programming education. Most studies were done three decades ago, under assumptions that miss important issues that today's students must confront. This paper presents rationale and details for a modernized study of plan composition that accommodates a broader range of programming languages and problem features. Our study design has two novelties: the problems require students to deal with data-processing challenges (such as noisy data), and the questions ask students to not only produce but also evaluate programs. We present preliminary results from using our study in multiple courses from different linguistic paradigms. We discuss several future studies that are prompted by these results.

CCS Concepts

•Social and professional topics → Model curricula; CS1;

Keywords: Plan composition, imperative programming, functional programming

1. INTRODUCTION

Most non-trivial programs encompass multiple tasks that collectively perform a computation. Computing the number of employees with low salaries, for example, involves searching a collection of employee records, determining which have low salaries, and counting the number of identified employees. A program that performs the computation could approach this problem in several ways: it could detect low salaries and count employees while traversing the employee collection, it could extract low-salary employees then count the results, it could first sort all employee salaries, and so on. Organizing the tasks of a problem into a program is called *plan composition* [9]. It is a key skill to develop in learning to program.

Plan composition is relevant whether or not a student plans to major in computer science or become a professional programmer. Many students seek just enough programming education to solve problems in other disciplines: for example, a biology student may wish to write scripts to process data from a research project. Such scripts often involve integrating multiple tasks. Deferring problem decomposition and plan composition to later CS courses is therefore not an option: we need effective pedagogies for this material in CS1 and non-major courses.

Existing plan-composition studies are dated by problem tasks, rubrics that failed to consider high-level variations in code structure, and implicit assumptions about the languages and constructs that students might use. Most studies (1) used imperative programming (2) with few data structures on (3) problems dominated by I/O. Today's students often learn different languages (functional, dynamic/scripting, OO), libraries, and a richer set of constructs for processing data. If teaching plan composition is important, we need to understand the impact of different languages, constructs, and pedagogies on how students structure code. We also need to modernize the study questions to encompass both linguistic differences and current computing contexts.

This paper makes three contributions in this direction. First (§ 2), we propose a concrete set of problems and questions that refine prior plan-composition studies. Second (§ 3), we present data from a preliminary, multi-national, multi-university study with these problems: three courses used functional programming and two used imperative programming. Third (§ 5), we identify additional study questions and design constraints that emerged from analyzing our data. These findings sharpen the goals of future plan-composition studies, while also suggesting an overarching framework for designing such studies.

2. STUDYING STUDENT PLANS

Prior planning studies focused analysis on the errors students made and the process by which students developed code [3, 12]. Our interests go farther: we want to identify factors that influence students' code structures, and the ways in which curricula, pedagogy, and programming language affect these factors. We therefore ask students to not only *produce* programs but also *assess* solutions (operationally, to provide a partial-order preference ranking of code we give that represents different plans). The latter helps us identify issues that students consider when thinking about program structure, a question with little prior research.

While prior planning studies featured simple console I/O, today's students program in a data-rich world. There are many operations—common at a high-level, different in their details—that students may need for processing data. Plan-

SIGCSE '16, March 02 - 05, 2016, Memphis, TN, USA
© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-3685-7/16/03…$15.00
DOI: http://dx.doi.org/10.1145/2839509.2844556

ning problems should exercise these operations. Concretely, we focus on three particular data-processing challenges: noisy data that should be *cleansed* prior to computation; flattened structured data that could be *reshaped*; and monolithic data from which relevant parts must be *extracted.* We also include problems over data that are *already structured* yet admit multiple interleavings of tasks.

We considered candidate problems from the computing education literature, technical papers, and course assignments. By design (to reduce bias), some authors are fans of each of functional and imperative programming. We rejected problems that struck any of us as "unnatural" (in our subjective opinions) for CS1 students learning our preferred style of programming; all authors approved of all selected problems. Ultimately, we chose six problems that include at least two samples of many of our data-facing criteria. We limited ourselves to six problems of relatively modest complexity because instructors would balk at adding a lengthy assignment to already-full classes.

We now present our proposed set of problems. In our preliminary study (§ 3), we gave the first three as programming problems. The latter three were given as ranking problems, with students provided correct, working code in the language of the course. Other partitions would also make sense, though we recommend putting one problem from each data-processing criterion into each activity. Students should do the programming problems first, to reduce bias from seeing the solutions in the ranking problems. Due to space limits, we present only summaries of the problem statements and solutions. The full, exact problem statements, as well as our coding rubric, can be found at cs.brown.edu/research/plt/dl/sigcse2016plancomp/

Programming Problems.

Palindrome A palindrome is a string with the same letters in each of forward and reverse order (ignoring capitalization). Design a program called `isPalindrome` that consumes a string and determines whether the string with all spaces and punctuation removed is a palindrome. Treat all non-alphanumeric characters as punctuation.

Sum Over Table Assume that we represent tables of numbers as lists of rows, where each row is itself a list of numbers. The rows may have different lengths. Design a program `sumLargest` that consumes a table of numbers and produces the sum of the largest item from each row. Assume that no row is empty.

Adding Machine Design a program called `addingMachine` that consumes a list of numbers and produces a list of the sums of each non-empty sublist separated by zeros. Ignore input elements that occur after the first occurrence of two consecutive zeros.

Ranking Problems.

For ranking, students were given multiple programs that solved each problem (10–20 lines per solution in functional code, a little longer in Java). Here we describe the code structure in prose. Our solutions are described in terms of helper functions, but the code could also be (and in Java, was) written using loops and variables that accumulate answers. The text marked with black bars (not given to students) summarizes the design decisions within each solution.

Rainfall Design a program called `rainfall` that consumes a list of real numbers representing daily rainfall readings. The list may contain the number -999 indicating the end of the data of interest. Produce the average of the non-negative values in the list up to the first -999 (if it shows up). There may be negative numbers other than -999 in the list (representing faulty readings). Assume that there is at least one non-negative number before -999. The solution options are:

A One function (or helper function) that iteratively accumulates three values: the remaining numbers, the total so far, and the number of days so far. When the numbers are exhausted, it returns the average using the other two parameters.

B One function that iteratively computes the sum of rainfalls. Another function that iteratively computes the number of days. Each one duplicates the cleansing logic. A main computation that uses these two functions, then computes the average.

C One function to cleanse the data. A main function that invokes the cleanser, then uses library or helper functions to compute the sum of rainfalls and number of days, and computes the average.

> Solution **A** uses a *single traversal* of the data, while the others use multiple traversals. We call solution **C** "clean first" and solution **B** "clean multiple".

Length of Triples Design the program `maxTripleLength` that consumes a list of strings and produces the length of the longest concatenation of three consecutive elements. Assume the input contains at least three strings. The solution options are:

A A helper function converts the flat list into a list of triples. The main function first uses this to generate triples of strings, then converts these to triples of numbers, then calls a function to find the maximum one.

B A helper function converts the flat list into a list of triples. The main function first converts the input into a list of numbers, then uses the helper to generate triples of numbers, then calls a function to find the maximum.

C A helper function iteratively accumulates the list of remaining strings, the maximum length so far, and the two previous lengths. It is then invoked on the appropriate arguments.

> Solutions **A** and **B** employ reshaping, and hence have multiple traversals of data. Each solution comes with a data structure (e.g., a class) for a single triple. Solution **C** uses a single traversal.

Shopping Cart An online clothing store applies discounts during checkout. A shopping cart is a list of the items being purchased. Each item has a name (a string like `"shoes"`) and a price (a real number like 12.50). Design a program called `checkout` that consumes a shopping cart and produces the total cost of the cart after applying the following two discounts:

1. if the cart contains at least 100 worth of shoes, take 20% off the cost of all shoes (match only items whose exact name is `"shoes"`)
2. if the cart contains at least two hats, take 10 off the total of the cart (match only items whose exact name is `"hat"`)

	Language	Country	Students	Timing
US1-IM	Java	USA	261 [70]	CS1-end
US1-FP	Racket, OCaml	USA	144 [42]	CS1-end
US2-FP	Racket	USA	106 [0]	CS1-end
FR-FP	OCaml	France	35 [18]	FP-end
GM-IM	Java or Python	Germany	80 [24]	CS2-start

Table 1: Students column reports total students and the number sampled when looking at code solutions.

The problem statement included specific data structures for representing items. The solution options are:

A The set of shoe entries is extracted, and their discount calculated. The set of hat entries is extracted, and their discount calculated. The total cost is calculated, and the two discounts applied.

B A helper function iteratively accumulates the cart contents, the total cost so far, the shoe costs, and the hat costs. When the cart is empty, the discounts are computed and applied to the total cost.

Solution **A** performs a separate traversal per discount. Solution **B** traverses the data only once.

Problem Characteristics.
Rainfall and *Palindrome* require cleansing. *Adding Machine* and *Length of Triples* could leverage reshaping, while *Sum Over Table* has clean, already-structured data. *Rainfall* and *Adding Machine* examine only a prefix of data. *Shopping Cart* suggests future extensibility (for additional discounts), and also illustrates the potential for *separation of concerns* [6]. Each admits a variety of solution structures.

3. PRELIMINARY STUDY

As a formative exercise, we used the study described in § 2 to explore differences across programming languages. We wanted both to check whether these six problems, and the combination of programming and ranking, were useful for eliciting cross-linguistic planning nuances, and to identify candidate differences for further study.

We asked students to provide code for the first three problems and to preference rank (ties allowed), with justification, solutions for the latter three problems. We gathered data from five courses (three functional and two imperative) across three countries. None was taught by the authors.

Table 1 summarizes the courses: names indicate the country and programming paradigm (functional vs imperative; the latter in either an OO or scripting context). The two US1 courses were from parallel CS1-CS2 sequences in the same department (both lead into the major). GM-IM was a Java-based CS2 course that students came to after CS1 in either Python or Java. FR-FP was a first course in functional programming for third-year engineering students. US2-FP was a functional CS1 course that only did the ranking problems (and hence had no sampled students).

Prior studies on Rainfall [3, 8, 9] suggest that imperative solutions typically traverse the input exactly once while functional ones often perform multiple traversals. These trends are consistent with each paradigm's programming patterns. We thus analyzed traversal preferences in both the programming and ranking problems. When sampling programming solutions for detailed analysis within each course, we selected an even distribution in how often students preferred single-traversal solutions in the ranking problems.

Logistics: US2-FP did the ranking problems only, in a 40 minute lab setting after 6 weeks of CS1. All other courses gave the task as a homework assignment with multiple days to solve it (the exact duration varied by course). In all cases, at least a majority of students wrote predominantly correct solutions, confirming that the problems were within their abilities and that the assessment below is of program structures that were not written in haste or under pressure (e.g., in a timed exam). US1-IM, US1-FP, and FR-FP gave the study as a homework assignment towards the end of the semester. GM-IM used similar conditions, but at the start of CS2. All students worked in a programming environment. We customized the problem wordings for each course to accommodate its default data structures (lists versus arrays), vocabulary ("program" versus "method"), and naming conventions (`isPalindrome` versus `is-palindrome`). Staff from each course reviewed the problems, both to fine-tune the customizations and to confirm that the problems were reasonable for the students. Two of the authors manually read all the responses and coded them according to the rubric linked in § 2.

3.1 Results: Programming Problems

We hypothesized that most imperative students would write single `for`-loops that interleaved all problem tasks, while the functional students would frequently handle some tasks in separate functions. Each of *Palindrome* and *Adding Machine* contradicted part of this hypothesis. *Adding Machine* also highlighted key nuances within single-traversal solutions. We discuss each question in turn.

***Palindrome*:** Our version of palindrome has three tasks: "skipping" punctuation and whitespace, ignoring capitalization (which we'll call "normalizing"), and checking the core palindrome property on the resulting characters. We clustered solutions by how they interleaved the first two tasks with the third: Mequal solutions handled both skipping and normalization before the main reverse-equality check; Mcase handled skipping before a main equality check that also performed normalization; Mskip handled normalization before a main equality check that also performed skipping; Mboth handled both skipping and normalization in the main equality check with a single traversal. (None reflects missing or unintelligible programs.)

	Mequal	Mcase	Mskip	Mboth	None
US1-IM	81%	7%	3%	0%	9%
US1-FP	56%	41%	0%	0%	2%
FR-FP	83%	0%	6%	11%	0%
GM-IM	71%	4%	8%	0%	17%

The table shows that Mequal, which performs the equality check in a *separate* traversal, was the most popular in every course. Across all four courses, only two students (both from FR-FP, a functional course) did single-traversal solutions. What explains the lack of single-traversal solutions, especially amongst the imperative students?

The answer is library functions for manipulating or comparing strings or lists (of characters). Many Java solutions used `replaceAll` or `toLowerCase`; Racket solutions often used a case-insensitive string-comparator. Primitives thus have significant influence on program structure. This may not sound surprising, but it has pedagogic consequences. Classic plan-composition papers [9, 12] focused on the integration of new statements into existing procedures. With primitives, the problem shifts to include *decomposition* of problems around tasks covered by primitives. Problem decomposition is a different skill than code composition.

***Adding Machine*:** Nearly every solution, whether functional or imperative, used a single traversal. Nevertheless, there were significant differences across the paradigms, some with pedagogic implications. Not surprisingly, functional students used recursion, while imperative students used `for`-loops. Within each pattern, solutions differed in which intermediate values they accumulated in parameters or top-level variables: the running sum of the current sublist, the output so far, both, or neither ("neither" can occur in a recursive solution that recurs on a new list with the running sublist sum in the first position). The following table summarizes solution structures based on the accumulated values. AccBoth plans accumulated both the sublist sum and the output so far. AccSum accumulated only the current sublist sum. Reshape created lists of sublists before computing the sums. Other solutions accumulated sublists or used nested loops (rather than top-level variables) for the sublist sums. None had no code relevant to the problem.

	AccBoth	AccSum	Reshape	Other	None
US1-IM	56%	0%	0%	27%	17%
US1-FP	10%	36%	14%	40%	0%
FR-FP	22%	17%	6%	33%	11%
GM-IM	48%	0%	9%	22%	22%

Solutions were more distributed in the functional courses. This likely reflects flexibility from having the input list be a parameter in recursive solutions: a solution could, for example, store the running sum in the first position of the list. `For`-loop solutions lack this flexibility. Only one imperative student (in US1-IM) used recursion: this solution passed a modified input list on the recursive call, but was missing too many tasks to reflect a clear structure.

Functional and imperative solutions seem prone to different errors:

- Several imperative students ran into trouble using the "enhanced" form of `for` (e.g., `for (i : anArray) ...`). In this form, it is impossible to determine the next element, which is essential for this problem (to check for two consecutive zeros). When trapped in this situation, many students inadvertently confused array contents with indices, e.g., comparing `i` with zero (rightly treating `i` as the content) but also comparing `i + 1` with zero (accidentally treating it as an index). (We conjecture that using the variable name `i`, which is usually associated with indices, makes this problem more likely, and suggest that users of enhanced for loops use a variable name more likely to suggest content than an index.)

- Students who used index-based `for` loops avoided this problem, but fell prey to (the usual) out-of-bounds errors when doing arithmetic on indices to detect consecutive zeros.

- Recursive solutions sit squarely between these two forms of loops: the recursive list is an inductive structure, so even though the program lacks index variables, it can still access subsequent elements of the list from the current head. Functional students who kept the running sublist sum in the first position of the list typically produced solutions that would fail on a non-empty sublist whose sum is zero.

- Imperative solutions that used the current sum to detect consecutive zeros fell prey to the same problem, but imperative students used a wider range of patterns for detecting consecutive zeros in the first place.

3.2 Results: Ranking Problems

For the ranking problems, we were primarily interested in whether the language students learned correlated with preferences for single-traversal solutions. § 2 outlines the structure of the solutions. We implemented each solution using typical structures from each language (e.g., we did not write recursive solutions in Java), but without primitives or operators that traversed data (such as built-in iterators or higher-order functions). We did not lead the students with particular issues to consider, but merely asked for free-form comments that justified their rankings.

The following table reports the percentage of *all* students (not sampled) who ranked the single-traversal solution as one of their top choices. (It is worth noting that even though ties were permitted, they never exceeded 15% for first-place choices, and were usually closer to 6–8%, indicating that students really did choose to express a preference.) On all three problems, students in the imperative courses had strong preferences for single-traversal solutions. Curiously, each functional course had a much stronger preference for single traversals on one problem, with a different problem preferred in each course. We do not yet have an explanation for this, but suspect it says something about curricular or pedagogic differences among the courses.

	Rainfall	*Length of Triples*	*Shopping Cart*
US1-IM	90%	93%	94%
US1-FP	71%	34%	48%
US2-FP	33%	33%	65%
FR-FP	60%	86%	43%
GM-IM	88%	81%	78%

Students' written justifications (sampled) cited issues such as runtime efficiency, readability, space consumption, code layout, separation of concerns, use of variables, lines of code, clarity of computations, and support for future code modification. No issue was raised consistently either within a course or within those who preferred a particular structure. Not surprisingly, however, efficiency was one of the top two cited issues (readability, in various forms, was the other). Some students raised it as a deciding factor (in favor of a single-traversal solution), but others cited it as a difference between solutions that was outweighed by another concern.

Three observations around efficiency comments warrant deeper investigation. First, in US1-IM, students often based decisions on efficiency, even though the professor insisted that he not only did not teach it, he expressly told his students to not consider it. Because most students in the class have no prior programming experience, their concern is unlikely to stem from their previous preparation. One possibility is that the large teaching assistant population (all undergraduate students who previously took the same course) is sending a different message that contradicts the professor's.

Another is that this is an inherent concern in how students think about computing—at least those that choose a Java-based course. We believe this warrants further investigation to understand the mindset of students.

Second, concern with efficiency and counting the number of traversals is not consistent with using library functions (such as `replaceAll` in *Palindrome*) that perform additional data traversals. In both imperative courses, roughly half of the students who used `replaceAll` also selected single loop solutions with justifications that cited efficiency. This suggests that students might have a weak cost model of library functions. There seem to be significant open questions here about what cost models students have, how they develop, and what role they should or do play in planning.

Third, several imperative students (16 of 59 in US1-IM, for example) criticized creating intermediate data as inefficient (each multi-traversal solution explicitly created such data). Creating intermediate data structures is common in functional programming, though less typical in imperative introductory courses. Again, however, library functions obscure intermediate data creation. In addition, in some languages it is *syntactically* much more concise to define an intermediate data structure than in others: creating the datatype and constructor for a flat triple of data needs only a single line in Racket, but at least 7 lines in Java. Built-in iterators (such as filters in functional programming) create intermediate data with a single operation; doing the same task manually requires several lines of code, which can make the program look more complicated. It is therefore unclear whether students' concerns about efficiency are truly about the run-time cost or actually statements about the syntactic appearance of code.

4. RELATED WORK

Historically, studies of program planning focused on how students interleaved I/O (including detecting sentinels and re-prompting on bad input) with computation. *Rainfall* is the classic planning problem [9, 10], with other I/O-focused problems proposed in other studies [11, 12]. Given that I/O mechanisms vary widely across models, problems with multiple tasks that arise from the problem data should be better suited to cross-course and cross-language studies.

In many early *Rainfall* studies [9, 11], students had not learned data structures or multiple forms of iterating over data. This vastly constrains the range of viable plans, leaving little to study beyond program errors [3, 8, 9, 13]. Students today learn a richer collection of constructs and libraries, thus letting us study relationships between the plans students choose and instructors' pedagogic decisions (which languages to use, primitives and patterns to teach, etc.) that influence students' choices.

Our project studies the relationship between programming languages (and their corresponding coding styles and pedagogies) and students' code structure. Ebrahimi [3] did the first cross-lingual *Rainfall* study in 1994. He compared rates of certain low-level errors across solutions written in four languages. He did not analyze high-level differences in solution structures (as we do), seemingly because the students had learned similar programming patterns across the languages [personal communication].

Fisler [4] studied *Rainfall* with functional programming students, finding that they produced diverse solutions, often correlated with use of library functions. One goal of our paper was to see whether those results generalized to problems other than *Rainfall*. This paper also adds the ranking questions component, which was not part of Fisler's work.

Theories of how people select plans and program structure [1, 7, 11] likely explain some of students' planning choices (such as which looping structures they use). These theories do not, however, fully account for design decisions such as whether to clean or parse data before processing it. Those decisions reflect general design principles. Ideally, studies should explore when and how students develop an appreciation for such design concerns.

The program structures that we consider here, based on cleaning and reshaping, as well as consideration of different kinds of iteration, represent programming patterns that one can teach explicitly. Ginat and Muller's work on Pattern-Oriented Instruction [5] shows that explicitly teaching patterns can improve student performance in problem decomposition and programming. Several (though not all) of the patterns that they teach correspond to standard primitives and higher-order operators from functional programming. Clancy and Linn [2] summarize approaches to teaching programming patterns, highlighting that such instruction must be multi-faceted, relating to both problem descriptions and a model of program execution. Their work may provide a lens for investigating differences across our study courses.

The ranking-problems portion of our study loosely relates to other work on code comprehension using *Rainfall*. Another part of Ebrahimi's study compared the correctness of students' *Rainfall* solutions with their performance on a code-comprehension task. Venables et al. [13] related student performance on writing programs including *Rainfall* with their ability to trace and explain programs with procedural loops. They found that students who could not trace accurately could not produce correct programs. Our study did not ask students directly to comprehend code; indeed, students could rank the problems based on surface features without deep understanding of how the programs worked. Nevertheless, a handful of students ruled out solutions believing them to be incorrect.

5. TOPICS FOR FUTURE STUDIES

Different programming languages embody different ways of structuring data and computations. We would expect the structure of students' code to vary somewhat across programming languages. This variance has not been studied sufficiently, especially with respect to skills like plan composition. Many early studies of student programming were done within single linguistic styles. Given the variety of languages and styles currently used for beginning students, the community needs a much deeper understanding of the interplay between core skills and linguistic factors.

This paper proposes a set of problems and a general framework for cross-linguistic studies of plan composition. The framework tries to identify parameters that can yield a robust set of problems for studying plan integration. Prior to our preliminary study, these parameters emphasized the nature of the tasks within programming problems. We took a data-processing approach, embracing challenges that many casual programmers are expected to confront: noisy data, flattened data, and selecting subsets of data. Our initial study suggests two additional parameters: (1) the existence (and student knowledge) of common library functions that align with some problem tasks, and (2) dependencies between data elements (such as relationships between consecutive elements) for some problem tasks. We posit that varying these parameters will deepen the value of planning studies.

Deeper analysis of library functions and language constructs should shed light on the aspects of different languages that matter for plan composition. For instance, Python is taught imperatively by some and functionally by others. We see such variation in the solutions students produce. Indeed, this challenges whether the typical paradigm classification of languages is even sensical.

We are also exploring students' views on program structure by having them assess code, not only produce it. Asking students to rank and comment on different solutions provides insight into the criteria that students are internalizing about "good" programs. We suspect these criteria are often not taught explicitly (as we saw with efficiency concerns). This kind of implicit knowledge acquisition is a vital but under-studied aspect of computing education.

Contrasting students' preferences in ranking solutions with the code they produced raised questions about the cost models that students have for library functions. We saw several students embrace primitives that traverse data multiple times when programming, only to castigate these same approaches as inefficient when ranking. Perhaps the conciseness of code written with primitives blinds students to their costs. Perhaps students cite efficiency because no other criterion comes to mind. A followup study should have some ranking problem solutions use primitives while others use equivalent computations written manually. This could help tease apart both students' cost models and whether, as we conjecture, concise code is perceived as more efficient.

Our data also reveal that some students criticize intermediate data for efficiency reasons, while others praise "thinking ahead" to other potential uses of such data (such as potentially reusing clean data from *Rainfall* in another, related, computation). We suspect that these reactions are learned from the problems students have seen in class. It is possible that simply showing imperative-programming students solutions that create intermediate data, even without directly discussing efficiency concerns, would alter perceptions that the inefficiency of such data is inherently bad.

With student populations changing, CS courses are increasingly called on to prepare a broader range of students to write not-too-complicated (but not trivial) scripts for data processing. For these students, teaching them to effectively use primitives and to write separate functions for separate tasks is likely less error prone than teaching them to interleave tasks into single-traversal functions. Put differently, our student population may call for shifting the goal of "plan composition" studies to look at how to teach *problem decomposition* as much as *code composition.* Classic plan-composition studies arguably emphasized the latter, at least as evidenced by evaluation metrics. Instead, we need to refine the rubrics that we use in planning studies to account for each of the decompositions of tasks around functions or traversals, errors within tasks, and errors in composing tasks and traversals. Our work with *Adding Machine* taught us that we also need to track the specific constructs through which students implement individual traversals. We similarly see value in tracking constructs in Python due to the different styles engendered by explicit loops versus constructing loops implicitly through comprehensions.

Should we deem this shift valuable, researchers should also query *instructors* about their ranking preferences and perceived value of efficiency. We have discussed different solutions to our study problems with multiple colleagues, many of whom teach introductory courses. Informally, we see faculty who work in imperative programming question solutions that perform multiple traversals (on the grounds of inefficiency), while faculty who program functionally raise principles (such as "clean data first" and "separate concerns") behind their multi-traversal solutions. Other projects, such as Ginat's pattern-oriented instruction [5] explicitly promote solutions that interleave tasks for sake of efficiency. A broader discussion among faculty as to what we will consider "well composed" programs may be in order to smooth existing differences in solutions across languages.

Finally, this study has refined our sense of what background information might help understand factors that influence planning. Our data suggest that meaningful "prior-knowledge" surveys should explore what datatypes students know, what iteration patterns they have learned over each, and which libraries they have used. This is more nuanced than "how many courses", or "what languages have you used", while targeting specific prior knowledge that might affect students' program structures.

Acknowledgments.

We thank Joe Beck, Amy Greenwald, Alan Schmitt, Peter Thiemann, and Andy van Dam. We appreciate Ardra Hren's feedback. Mike Clancy presented us the adding machine problem. This work is partially funded by the US NSF.

6. REFERENCES

[1] J. R. Anderson, R. Farrell, and R. Sauers. Learning to program in LISP. *Cognitive Science*, 8:87–129, 1984.

[2] M. J. Clancy and M. C. Linn. Patterns and pedagogy. *SIGCSE Bulletin*, 31(1):37–42, Mar. 1999.

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

[4] K. Fisler. The recurring rainfall problem. In *Proceedings of ICER*, 2014.

[5] O. Muller, B. Haberman, and D. Ginat. Pattern-oriented instruction and its influence on problem decomposition and solution construction. In *Proceedings of ITiCSE*, 2007.

[6] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, Dec. 1972.

[7] P. L. Pirolli. *Problem Solving by Analogy and Skill Acquisition in the Domain of Programming.* PhD thesis, Carnegie Mellon University, Department of Cognitive Psychology, 1985.

[8] Simon. Soloway's Rainfall problem has become harder. *Learning and Teaching in Computing and Engineering*, pages 130–135, 2013.

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

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

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

[12] J. C. Spohrer and E. Soloway. Simulating student programmers. In *International Joint Conference on Artificial Intelligence*, pages 543–549, 1989.

[13] A. Venables, G. Tan, and R. Lister. A closer look at tracing, explaining and code writing skills in the novice programmer. In *Proceedings of ICER*, pages 117–128, 2009.