The Recurring Rainfall Problem

Kathi Fisler
WPI Dept of Computer Science
kfisler@cs.wpi.edu

ABSTRACT

Many studies have used Soloway's Rainfall problem to explore plan composition and programming errors by novice programmers. Few of these have explored students from CS1 courses that use functional programming. The concepts and programming styles commonly taught in such courses give CS1 students more viable plan-composition options than in traditional imperative CS1 courses. Using data from five functional-language CS1 courses at four schools, we show that our students choose different high-level structures and make fewer low-level errors compared to results of other Rainfall studies. We discuss the potential role of language in these results and raise various questions that could further explore these effects.

Categories and Subject Descriptors: K.3.2 [Computers and Education]: Computer and Information Science Education

Keywords: Plan composition, novice programmers, functional programming

1. INTRODUCTION

Soloway's Rainfall problem [10] has been used in several studies to assess students' progress in learning to construct programs. Rainfall is interesting because it is conceptually straightforward (a variation on averaging a collection of numbers), yet with enough components to raise subtleties in code. Soloway proposed the problem in the context of exploring *plan composition*: how students weave together the different components of a problem into a single program.

Over the years, multiple papers have reported on student performance on Rainfall, sometimes categorizing the errors that students make. Across prior studies, students typically solved Rainfall under three common constraints: they (nearly always) programmed imperatively, they (usually) obtained the numeric data through keyboard input, and they (often) had limited prior exposure to data structures, with the possible exception of arrays. These constraints don't arise in CS1 courses based on functional programming. Such courses emphasize different linguistic constructs (e.g., recursion instead of loops), rarely cover interactive I/O, and have students work extensively with lists from the early weeks of a course. This naturally raises a question: what do students from functional-first CS1 courses do with Rainfall?

Both facility with lists and limited use of I/O are interesting parameters. Because lists are easy to construct dynamically (unlike many array implementations), they enable a variety of viable high-level structures for Rainfall programs; some of these are harder or inaccessible with other or no data structures. Earlier Rainfall studies have shown that I/O patterns are difficult to get right, especially when data can be noisy. Avoiding I/O frees students to focus on computational tasks that arise across devices and platforms. Studying Rainfall on functional-first CS1 students could change how we view this classic question.

This paper makes three contributions. First, we propose a two-level analysis methodology for plan composition that looks at (a) the high-level structure of a solution, and (b) low-level errors when implementing tasks within that structure. While our coding rubric draws heavily on those from prior Rainfall studies, we found existing rubrics too coarse-grained for important nuances that arise in our dataset. Second, we apply our methodology to over 200 Rainfall samples across five functional-first CS1 courses at four schools (three universities and one high school) in the USA; we partly explore an additional dataset from a traditional Java-based CS1. Our functional-first participants produce solutions with rather different structures than in earlier studies, and seem to make fewer low-level errors. Our third contribution reflects the formative nature of this study: we identify several open research questions inspired by our observations.

2. THE RAINFALL PROBLEM

Soloway proposed the Rainfall problem in the 1980s [10]. The original wording was simply about computing averages:

> Write a program that will read in integers and output their average. Stop reading when the value 99999 is input.

Soloway identified four key goals within this problem: taking in input, summing the inputs, computing the average (which involves counting the inputs), and outputting the average. Programs that solve this problem must compose fragments of code that achieve each of these goals. Soloway proposed this problem in part to study approaches to plan composition, particularly in novices.

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 ACM 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.

ICER'14, August 11-13, 2014, Glasgow, Scotland, UK.
Copyright 2014 ACM 978-1-4503-2755-8/14/08 ...$15.00.
http://dx.doi.org/10.1145/2632320.2632346.

The "Rainfall" context has inspired many variations on the problem: some versions [1, 9] added negative numbers that the programmer should treat as input errors; some added output requirements [1], such as reporting the count of days with non-zero rainfall or the maximum daily rainfall in addition to the average. All variations shared common core goals of summing, counting, averaging, input and output.

In this study, we include negative numbers in the input, but only require producing the average as output. We also provide the inputs in a data structure (as our students had not learned I/O). Our specific wording is as follows:

> Design a program called rainfall that consumes a list of numbers representing daily rainfall amounts as entered by a user. 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.

3. A FUNCTIONAL-FIRST CS1

Functional programming has many broad characteristics, including using recursion over unbounded-depth data structures (such as lists and trees), using few (if any) side-effects, and parameterizing functions over other functions. Functional languages and their corresponding programming styles vary widely (e.g., contrast LISP, Haskell, and Erlang).

This study focuses on a particular functional-first CS1 curriculum, *How to Design Programs* (henceforth HTDP) [2]. HTDP teaches a data- and test-driven approach to program design. While HTDP can be (and has been) used with non-functional languages, the textbook and accompanying software use Racket (a variant of Scheme).

The first part of HTDP teaches a bottom-up method for designing programs: given a problem, students define required datatypes, make concrete instances of the data, write concrete test cases (both inputs and expected outputs), write a code skeleton that traverses the input structurally (but does no problem-specific computation), then complete the skeleton with problem-specific code. Each step typically counts in grading: functions that produce correct answers but whose code structure deviates from the shape of the input data, for example, lose significant points (in Soloway's terms [10], HTDP requires code to have an *explanation*). HTDP takes students through this same core method multiple times on increasingly rich data structures: atomic values, structures, nested structures, lists, and (in most college-level courses for CS majors) trees. Later parts cover top-down design and problems requiring more than structural recursion.

HTDP teaches a limited set of language constructs: function definition, function invocation, conditionals, user-defined structures, lists (built-in), and perhaps naming intermediate computations. Roughly 5 weeks into a college-level CS1 for majors, students learn higher-order functions (such as `filter` and `map`) that implement common iteration patterns over lists, taking a function over individual list elements as input. In contrast to most non-functional CS1 courses, students do *not* learn standard `for/while` loops, arrays, or I/O; assignment operators, if covered at all, come towards the end of the course, and then only for use in contexts that require memory across calls to the same function.

HTDP's emphasis on a structured recipe for design makes it a particularly interesting context in which to study plan composition. Two aspects of this recipe—writing tests first and writing code against skeletons that traverse data—are particularly relevant. The testing emphasis means that students should have thought out (and written down!) the space of program inputs before writing even a single line of code. A good set of test cases would capture many common troublespots (such as the possibility of an empty list of inputs). The emphasis on skeletons that match the shape of data (identical to the Interpreter pattern in Object-Oriented design [3]) prescribes a default plan. As such, these skeletons are schemas in Soloway's sense, but aligned to data rather than control. HTDP seeks to have students master this bottom-up approach before tackling problems that require top-down planning. As a result, problems (like Rainfall) that require at least some top-down planning are fascinating testbeds for HTDP.

4. PLAN COMPOSITIONS

Rainfall requires students to compose code implementing multiple tasks. Our version (in Section 2) requires six tasks:

- Sentinel: Ignore inputs after the sentinel value
- Negative: Ignore negative inputs
- Sum: Total the non-negative inputs
- Count: Count the non-negative inputs
- DivZero: Guard against division by zero
- Average: Average the non-negative inputs

Our version omits I/O tasks common to earlier formalizations of Rainfall (as our students had not studied it). Of course, this omission changes the problem and some of its standard plan-composition nuances (such as where a student should re-query a user for an invalid input). Although Simon [9] questions whether such changes alter the problem too much, we believe the fundamental questions about plan composition remain unchanged.

Conceptually, these tasks can be executed and combined in various ways. Figure 1 shows three (of many) examples:

- "Single Loop" (left) iteratively consumes inputs, increments sum and count on each non-negative input, then checks for division by zero and computes the average upon reaching the sentinel.
- "Clean First" (middle) iteratively adds non-negative inputs to a data structure until the sentinel is reached. It then counts and sums the cleaned data, checks for division by zero, and computes the average.
- "Clean Multiple" (right) traverses the input twice, once to count non-negative inputs and once to sum them. Each traversal terminates at the sentinel. It then checks for division by zero and computes the average.

The data structures that students know affect which options are feasible for them. A student who has learned no data structures (an apparent constraint in early plan-composition papers) has only the "Single Loop" option, as the other two require building intermediate data structures. A student who has learned only arrays could implement one of "Clean First" or "Clean Multiple", but probably would not because a typical array declaration expects the programmer to provide a size, which is unknown (the technique of creating a larger-than-needed array is not typically covered in

```
   ** SINGLE LOOP **
repeat until find sentinel {
  if next input is non-negative
    increment count
    add it to the running sum
}
if count is at least 1
  compute the average as sum/count
else report ‘‘no data’’
```

```
    ** CLEAN FIRST **
repeat until find sentinel {
  if next input is non-negative
    add to set of clean data
}
if at least one clean datum
  count the clean data
  sum the clean data
  compute average as sum/count
else report ‘‘no data’’
```

```
   ** CLEAN MULTIPLE **
repeat until find sentinel {
  if next input is non-negative
    increment count
}
repeat until find sentinel {
  if next input is non-negative
    add it to the running sum
}
if count is at least 1
  compute average as sum/count
else report ‘‘no data’’
```

Figure 1: Three high-level structures for Rainfall. We present these in imperative pseudocode for the benefit of readers who are less familiar with functional programming.

CS1). A student who has worked with lists, however, has all three options readily at her disposal.

Going into this study, we hypothesized that many students trained in list-based functional programming would use a high-level structure that traversed the inputs more than once. HTDP drills students on the idea of ”one task per function”, emphasizing correctness over performance; this might dispose students to one of the multi-traversal plans.

5. METHODOLOGY

5.1 Data Collection

We collected data in the Fall of 2013. We contacted colleagues teaching HTDP-based CS1 courses at two institutions, then mentioned the study on a mailing list for users of HTDP. Three additional faculty (two university, one high school) offered to participate.[1]

All sites used the problem wording given in Section 2. The logistics of assigning the problem differed across the courses: some included the problem on exams, others used supervised lab time, and others used out-of-class time (but still required the assignment). Table 1 summarizes the nature of the courses, sample sizes, and the conditions under which students completed the problem in each course.

The solutions were passed along to the author either electronically or on paper via postal mail, as appropriate. Solutions were stripped of identifying information before being supplied to the author. Resource constraints led us to code only a subset of the solutions from some courses. We coded a higher percentage of the T3Non samples since their students were both non-majors and had the least experience with HTDP. We coded all HS and T1Acc samples since their populations were smaller. We used the `sample` command in R (ver 2.15.2) to randomly choose which samples to code. In hindsight, this method did not guarantee that each sample was representative within the course. The author did all of the coding, so there was no need to assess inter-coder reliability. All statistical analysis was done in R (ver 3.0.3).

5.2 Coding Composition Structures

We coded composition structures by reducing each student's program to a regular-expression-like summary. We used a single-letter token for each of the six tasks (S for sum, T for sentinel, etc), combined with operators indicating interleaved (&), parallel (|), sequential (;), or guarded (→) composition. As examples, the following two codes capture the “Clean First” and “Single Loop” programs from Figure 1:

```
Clean First: T&N ; D -> (S|C); A
Single Loop: (T & N & S & C) ; D -> A
```

These expressions capture the structural essence of each solution. They do not capture details such as how many functions students wrote, whether sequential tasks were in the same or different functions, or whether students used higher-order functions. Additional coding captured some of this information. For each task within each solution, we recorded when the task was implemented (a) within the main Rainfall function, (b) within a helper function, (c) across multiple functions, or (d) missing entirely. We also recorded whether the task was implemented using a built-in operation or higher-order function (e.g., `length` or `filter`).

Some solutions were sufficiently mangled that we could not construct meaningful summaries for them. In such cases, our code either captured which tasks were represented at all, or recorded “None” if no tasks were evident.

Overall, our dataset included 32 different high-level structures that included all six required tasks. For analysis, we grouped these into six clusters, as discussed in Section 6.1.

5.3 Coding Task Errors

When a student implemented a task incorrectly, we used separate codes to record the errors. Our error codes derive (with minimal changes) from Ebrahimi's [1] Rainfall codes, which in turn built on Ostrand and Weyuker's error-coding system [8]. An error code is a sequence of 2 or 3 letters: a category of difference, a component of difference, and (optionally) the extent of difference (which defaulted to Full if omitted). Each piece had the following options (normal-face items are from Ebrahimi; italics mark our additions):

Categories of Difference: Missing (X), Misplaced (P), Malformed (F), Spurious (S), *Inconsistent with Tests* (I)

Component of Difference: Initialize (I), *Base Case* (B), Update (U), Guard (G), *Input* (P), *Header* (H)

Extent of Difference [optional]: Partial (P), Full (F)

Thus, an error code of F-B means “malformed base case” (which might arise if the base case of a recursive function

[1] None of the data were collected in the author's own school. The author is, however, a long-time HTDP instructor.

Label	Course Type	School Type	Samples	# Coded	Conditions	Language
T2	CS1	2nd Tier	224	63	Exam, 20-30 mins, 10 weeks in	Racket
T1	CS1	1st Tier	154	61	Home, no limit, 13 weeks in	Racket, OCaml
HS	CS1	High School	7	7	Lab, 20 min, 18 weeks in	Racket
T3Non	Non-majors Intro	3rd Tier	65	43	Lab, 10 mins, 10 weeks in	Racket
T1Acc	Accelerated CS1	1st Tier	44	44	Home, no limit, 13 weeks in	Racket, Pyret

Table 1: Summary of participating courses (all in the USA). "Tier" in the School Type column reflects competitiveness of admission (lower numbers are more competitive); this approximates general academic ability. The two 1st-Tier courses are from the same university, with T1Acc being an accelerated course that students tested into after a month in T1 (most in T1Acc had prior programming experience from high-school). For the two populations that worked at home, nearly all students self-reported spending no more than 30 minutes on the problem.)

did the wrong test or had a serious syntax error), while X-G-P means "missing guard partially" (which might arise if a student guarded against division-by-zero on only some control-flow paths through a program).[2]

Three of Ebrahimi's original components of difference—input, output, and loop—can not arise in our programs. We folded his "syntax concept" errors into "malformed", as both manifest similarly in Racket.

Nuances in our data required options (italicized) beyond Ebrahimi's. *Inconsistent with Tests* applies when a student's test case expected a different result than his program produced. *Base Case* captures errors in setting up the base case for a recursive function; if the base case has the right guard but the wrong return value, we use the (existing) Initialize code. *Header* captures problems in the name or parameter list in a function definition. *Input* captures problems in the actual parameters passed to functions (such as a task being correct relative to its formal parameter, but receiving an incorrect actual parameter). *Extent of difference* captures cases in which a task was implemented correctly on some control-flow paths but not others (e.g., forgetting to check for division-by-zero after cleaning negatives from the input).

In contrast to earlier studies, we recorded errors at the level of individual tasks (Sum, Average, etc), rather than for the entire program. This granularity seems important: students might make some errors (such as forgetting guards) more frequently in some tasks. In functional languages, where students might write separate functions for individual tasks, per-task coding helps cluster errors that occurred in the same function. The downside is that the same source-code error may affect (and thus count towards) multiple tasks, thus inflating sums of errors across tasks.

6. DATA ANALYSIS

We first analyze high-level program structure (methodology from Section 5.2), which reflects the essence of plan composition. We then explore low-level errors within those structures (methodology from Section 5.3). Finally, we contrast our results to those of prior Rainfall studies as best as we can around our subtle differences in methodology.

6.1 High-Level Composition Structure

Table 2 summarizes the high-level structures that students used, broken out by course. We have six clusters of high-level structure: the three from Figure 1 ("Single Loop", "Clean First", and "Clean Multiple"), "Clean After" (attempt to adjust the results of sum and count to handle negative and sentinel), "No Cleaning" (omitted negative and sentinel, but included sum/count/average), and "Unclear" (generally mangled code with no clear structure).

	T2	T1	HS	T3Non	T1Acc	total
Single Loop	1	23	0	1	13	38
Clean First	40	25	1	1	20	87
Clean Multiple	12	13	3	15	9	43
Clean After	1	0	1	1	0	3
No Cleaning	1	0	1	5	0	7
Unclear	8	0	0	19	2	29

Table 2: Number of students implementing each high-level structure within each course.

Two observations stand out from this table: the distribution of solutions across different high-level structures, and the low percentage but skewed distribution of students producing "Unclear" solutions. We discuss each in turn.

Diverse Composition Structures.

Table 2 shows that the single-loop structure of traditional `for/while` loop solutions was not dominant in our data. This is partly due to course organization: neither T3Non nor HS had covered recursive functions that accumulate answers in parameters (an advanced topic in HTDP, coming after trees and higher-order functions); T2 started the topic just before the exam that included Rainfall, but most sections were told not to use it on the exam. Even in courses where students had either seen this material (T1) or learned **for/while** loops in high-school (most students in T1Acc), other structures are more common than "Single Loop".

What drives students towards "Clean First" or "Clean Multiple" instead of "Single Loop"? Spohrer and Soloway's model of plan composition in MARCEL [11] suggested that novices start from an initial plan, refining it as needed to add tasks and handle errors. If this theory holds, we should consider how different initial plan choices might lead to different structures. For example, students might:

- Start with the overall problem (average), which most students know is computed from the sum and length of a list. Sum and length are common exercises in functional courses: students could either augment their existing plans for these to also clean input ("Clean Mul-

[2] The full coding manual is available at `www.cs.wpi.edu/~kfisler/Pubs/icer14-rainfall/`.

tiple") or clean data before using their existing plans ("Clean First").

- Start by asking which tasks higher-order functions could capture succinctly. Sum, Count, and Negative are often used to illustrate higher-order functions. Students could clean input ("Clean First") to provide data to their existing higher-order plans for these tasks.
- Start by traversing the inputs, expecting to process each as they go ("Single Loop").

We are not claiming that our students did follow these thought processes (our data cannot indicate this). Rather, we are claiming that these are initial plans that (a) are accessible to our students and (b) have plausible paths to each of the common structures.

The idea of built-in or higher-order functions as initial plans warrants more discussion. Soloway described plans as "template-like structures" for common problems [10]; higher-order functions merely turn these templates into functions that are parameterized over other computation. For example, `filter` takes a list and a predicate over individual list elements, returning the sublist of elements that satisfy the predicate. The names of higher-order functions indicate their effects on the input: a programmer who sees `(filter <Foo> numList)` knows immediately that the result is a subset of `numList` chosen by `Foo`. Conventional loops lack this feature. Seeing `for` simply says that something will be traversed, but nothing about what will happen during that traversal. In this sense, higher-order functions may provide more concise and targeted plans than conventional loops. This echoes Miller's proposal for looping "macros" whose names align with functionality, which arose after studying program plan composition through natural-language [6].

Students in T2, T1, and T1Acc had seen higher-order functions prior to this study; all students had seen `length` (of a list). The following table summarizes how many students in each high-level structure category used higher-order or built-in functions for each task. It shows that students with "Clean First" solutions use these features heavily.

	N	**Sum**	**Count**	**Neg**	**Sentl**
Single Loop	38	1	1	0	1
Clean First	87	72	83	24	3
Clean Multiple	43	4	2	2	3
Other	39	1	12	0	1

# of students using built-in functions, by structure

Count was nearly always implemented with the `length` function (built-in) on lists (a few students used a higher-order function called `fold`). Even in T3Non, which had a high rate of "Unclear" structures, 11 students used `length`. Within "Clean First", all but two of the Sum cases used `fold`; all but three of the Negative cases used `filter`. The "Single Loop" entries came from one student who wrote all of Rainfall via a single use of `fold`.

Take Away: The overwhelming correlation between high-level structure and using built-in and higher-order functions in "Clean First" suggests that knowing these functions significantly influences how students structure Rainfall solutions.

Open Questions: Do students who know higher-order functions use them as initial plans? Are looping constructs whose names describe impacts on data more useful as initial plans than those named more for control-flow?

Unclear Composition Structures.

Only 14% of students (31 out of 218) had "Unclear" composition structures. This includes 7 students with no relevant code, and another 5 who had a function for average but did not integrate it with the other tasks. Thus, only 5% of students (12 of 218) had no evident plan for solving Rainfall.

All but one of the remaining 19 students (8% of total) used the HTDP design recipe to write Rainfall: they wrote a recursive function with a single parameter (for the list), and attempted to compute the average piecewise as they encountered each element. These solutions had the following structure, with some mix of the Sentinel, Negative, Sum, and Count tasks intermingled in place of the ellipses:

```
(define (rainfall alon)
  (cond [(empty? alon) 0]
        [(cons? alon) ... (first alon)
                      ... (rainfall (rest alon))]))
```

These students' main failing lay in using the default HTDP list skeleton as their overall plan; this plan was not suitable for Rainfall. This is not an indictment of HTDP per se: these students may perform well on writing straightforward traversals. In addition, the parts of HTDP that address accumulating data and problem decomposition occur later in the course, beyond what some of our participating courses had covered. The distribution of "Unclear" solutions across courses is therefore relevant.

Ten of the 18 students who naively followed HTDP were in course T3Non, which had three potentially-confounding variables: these students had been given only 10 minutes to work on the problem, they had covered less material than those in other courses, and they were non-majors. If a T3Non student had tried the default and realized another approach was needed, she would not have had time to implement it. We therefore interpret the T3Non data as indicating students initial plan choices (ala Spohrer and Soloway): that many of the "Unclear" structure students at least ended up in the HTDP default suggests that HTDP students are learning techniques for getting started on programs.

Take Away: HTDP students absorb techniques for writing list traversals, but likely need later parts of the course to handle situations when structural recursion does not apply.

Open Questions: What is the relationship between extent of coverage of HTDP and performance on Rainfall?

Which High-Level Structure is Preferable?.

Are certain high-level Rainfall structures preferable for students coming out of CS1? "Single Loop" appears most efficient (traversing the input only once and not building intermediate data);[3] "Clean First" separates the concerns (tasks) of the problem, which simplifies maintenance (and arguably human comprehension).

We informally polled long-standing HTDP instructors for their preference among "Clean First", "Clean Multiple", and "Single Loop". We gave them a canonical solution in each structure (all written in Racket), and asked which they preferred and why. We got 8 responses, 3 from instructors who had provided data for this study. Uniformly, they preferred

[3] The efficiency argument is not valid in light of long-standing compilation techniques that interweave nested list traversals (e.g., [13]), though belief in it persists.

either "Clean First" or "Clean Multiple" to "Single Loop", with justifications referring to separation of concerns, ease of testing individual tasks, and overall cleanliness.[4] Half preferred "Single Loop" to "Clean Multiple" on grounds of efficiency (with both behind "Clean First").

Open Questions: What design principles (e.g., clean data first, reduce traversals, etc) should students prioritize coming out of CS1? Which principles should be reflected as plans that students choose instinctively? Which should students treat as optimizations to apply once programs are working?

6.2 What Errors Do Students Make?

Low-level programming errors occur both within implementations of specific tasks and in the glue that composes them. Arguably, many student errors in previous plan-composition studies have been in the glue: for example, students might know how to increment a counter, but perform the increment in the wrong location. In functional programming, individual functions tend to align with individual or a few thematically-related tasks. Some mistakes are also harder to make in functional languages: when variables are parameters to recursive functions, it is harder to miss updating them. We might expect this isolation of task to lead to fewer low-level implementation errors, both within and across functions. This section explores low-level errors.

6.2.1 Error Rates Per Task

The most frequent error in our data, by far, was omitting a task entirely. The *percentage* of students omitting each task within each high-level structure category is as follows:

	Sum	Count	Neg	Sentl	DivZ	Avg
Single Loop	0	0	16	0	16	0
Clean First	0	1	18	3	36	0
Clean Mult	0	0	58	0	52	0
Clean After	0	0	100	0	33	0
No Cleaning	14	14	100	100	43	0
Unclear	24	66	97	41	59	38

% of students within category who omitted task

"No Cleaning" is defined by omitting both Negative and Sentinel, so their 100% omission rate is not surprising. Students with a clear structure rarely omitted Sum, Count, Sentinel, or Average. Many omitted Negative and DivZero, though these are rather different omissions: failure to include Negative means that students *missed part of the overall problem statement*, whereas failure to include DivZero means that students *failed to consider a subset of possible inputs to the problem.* Different techniques are likely needed to mitigate each problem, which raises another question:

Open Questions: Would asking students to provide test cases as part of plan-composition studies help identify the source of certain classes of errors?

The relative rates of omission of Negative and DivZero across "Single Loop", "Clean First", and "Clean Multiple" suggests that perhaps students in one of the first two categories are simply stronger programmers. This is plausible, as we have already argued that "Single Loop" is an advanced approach under htdp. Course-level data does not, however, support this hypothesis: students in (the accelerated) T1Acc make significantly more errors of omission than their fellow students in T1 on many cells of this table.

Setting aside missing tasks, we now explore the low-level errors that students made within (at least partially) implemented tasks. Our error-coding method (Section 5.3) has 30 possible error categories (setting aside the "partial" modifier), of which 21 appear in our data. For sake of space, here we report in detail only on errors in solutions with a clear high-level structure (N=187) that occurred in more than 5% (10) of the solutions across all tasks. The following table lists the number of occurrences of each error code within each task. Since the same error in source code could affect multiple tasks, the total within each row may be larger than the number of students who made the corresponding error.

	Sum	Count	Neg	Sentl	DivZero	Avg
X-G-P	0	0	1	0	33	0
X-B	12	6	2	6	0	0
F-U	7	5	4	6	1	3
F-G	2	2	11	5	1	0

Error count per task, clear structure only (N=187)

The 33 X-G-P (missing guard partial) DivZero errors reflect a student checking emptiness of the original input list but forgetting this check *after* handling negatives and the sentinel. Half of these cases were in "Clean First"; the other half split between "Clean Multiple" and "Single Loop".

Of the 26 X-B (missing base case) errors, 19 occur with "Clean Multiple". By definition, such solutions implement multiple tasks in one function, so a missing base case might affect more than one task. A total of 10 students made the 19 X-B errors within "Clean Multiple" solutions. On the one hand, these errors are surprising within htdp, as students who start from the list skeleton would automatically (and instinctively) include the base case. However, the low rate of this problem (under 2% of students) suggests that perhaps the htdp list skeleton is doing its job.[5]

The F-G (malformed guard) errors in Negative often reflect treating 0 as negative (e.g., using $<$ instead of $\leq$).

Overall, our data show fairly low error rates once students have a clear high-level structure. Setting aside missing tasks and solutions with "Unclear" structures, we found a total of 153 coded errors, some of which are over-counted (as discussed in Section 5.3). A total of 98 students had at least one error (other than "missing"). Thus, our data set has between 98 and 153 unique error locations within 187 solutions with clear high-level structures. An error rate below one per solution is quite low relative to other Rainfall studies. Including missing tasks as errors yields 296 errors within clear structures, for a maximum error rate of 1.58 per student.

Low-level error counts varied only slightly with high-level structure. The following table summarizes the total number of errors per student within the three major high-level structures. Each cell indicates the percentage of students with the given structure (row) and error count (column). Some errors are over-counted in "Clean Multiple". Since X-G-P errors in DivZero were the most common overall, the last column reports the percentage of students who made that error. The differences between "Single Loop" and "Clean Multiple" are not significant, though they become so (p =.002) without X-G-P. We do not compare significance of these

[4] A couple of T1Acc students left apology-like comments in their code for choosing "Single Loop" for efficiency reasons.

[5] We cannot compare data on this error to previous studies, as no prior studies included the X-B code.

against "Clean Multiple" given its over-counting. This table supports our claim that "Single Loop" solutions came from stronger programmers.

	0 errors	**1**	**2**	**3-6**	**9**	X-G-P
Single Loop	74	21	5	0	0	18
Clean First	66	22	7	6	0	18
Clean Multiple	46	23	13	15	2	21

% of students with each error count, by structure

Finally, our error codes distinguish within-task errors from between-task errors: the latter correspond to mistakes in composing functions for separate tasks. We find only 25 instances of between-task error codes (F-P, X-P, and P-P) in the entire data set. Of these, 23 were malformed actual parameters (F-P), 18 of which occurred in solutions with "Unclear" structure. We had only 3 instances of between-task composition errors in solutions with clear structure.

TAKE AWAY: HTDP solutions with clear high-level structure have low error rates and few composition errors.

OPEN QUESTIONS: Can HTDP and top-down planning curricula (such as Pattern-Oriented Instruction [7]) complement one other to improve performance on Rainfall?

6.3 Comparison to Other Rainfall Studies

Two prior Rainfall studies had conditions that overlap our own: Ebrahimi [1] included some functional programming, while Simon [9] provided the input in a data structure rather than require I/O. This section relates our findings to theirs, as best we can within the limits of differing methodologies.

Comparison to Ebrahimi.

Ebrahimi [1] studied student performance on Rainfall in four languages (C, FORTRAN, Pascal, and LISP). It is tempting to compare our error rates to those of his LISP and non-LISP students. Unfortunately, such comparison would be imprecise at best. Although our error codes are derived from his, our two-level methodology captures missing tasks outside of error codes, while he captured them within. Unlike Ebrahimi, we did not record low-level error codes for students whose solutions had no relevant code. Both of these undercount certain errors, while our per-task error-coding method overcounts others. Although these differences prevent direct comparison, differences in magnitude of our error rates suggest that our students did better. The most frequent errors among Ebrahimi's 20 LISP students were missing if-guards (50%; includes DivZero), misplaced updates (40%, includes computing average in loop and sum/count outside of loop), malformed updates (40%), and spurious output (40%, not relevant for our data). These rates are noticeably higher than for any of our error codes.

Differing course styles also hinder meaningful comparison of our data with Ebrahimi's. His students were in a position to write imperative LISP code: they had seen assignment operators, I/O, and iteration, and had previously used Pascal. They had been "encouraged" to use recursion and functions for each task when writing Rainfall.[6] We do not, however, know whether students heeded these instructions. Code written in functional languages is not necessarily "functional" (e.g., it might use side-effects): this is an important caveat for any study that crosses linguistic paradigms.

Comparison to Simon.

Simon [9] reported broad categories of mistakes and implementation choices across 149 students in a C#-based CS1. Four of his categories do not apply to us: using a `while` loop, using a `for` loop, initializing sum, and avoiding integer division.[7] The rest reflect at least partial implementation of specific tasks. In the following comparison, we report ranges that cover best- and worst-case interpretations of Simon's grading in our context. Ideally, percentages should be low in the first row and high in the others.

Simon Issue	Simon %	Our % range
No (pertinent) code	36	3 – 6
Loop terminated by sentinel	22	81 – 90
Days counted within loop	21	85 – 89
Negative values ignored	40	57
Guarded division for average	0	55 – 61

For "no (pertinent) code", we report a range from students with no discernible plan (low) to students with either no plan or one marked as "Far Off" (high). For the remaining rows, we report the range of students who included each task at least partially (high) or did so within a clear high-level structure (low; guarantees that a student had a recursive function—a.k.a. loop—covering that task).

Our students are clearly performing better than Simon's. Given that he also avoided I/O, this comparison suggests that that the difficulties with Rainfall go beyond I/O plans. Our requirements on negative numbers were slightly different (ours dropped them while his treated them as zero), but this seems insignificant as both versions require checking the sign of each input. Thus, we have to wonder whether the style of programming taught in our respective versions explains the large differences in performance. Additional studies would be required to explore this further.

Comparison to a Java-based CS1.

One university instructor on the HTDP listserv collected data in his (non-HTDP) Java-based CS1 course (which had covered arrays, but not lists). He used our problem wording, substituting "array of integers" for "list of numbers". We did not code his data (N=51) for low-level errors. Coding for high-level structure showed 50 "Single Loop" solutions: 28 used `for`, 12 used `while`, and 8 nested a `while` directly within a `for`. Only 1 student had two loops in sequence (`while` to clean the data then `for` to compute the average); only 2 students (including this one) used a second array. This supports our claim that something about either functional programming or HTDP steers students towards non-"Single Loop" solutions with auxiliary data structures.

7. DISCUSSION

In 2010, Mark Guzdial [5] challenged computing education researchers to make progress on Rainfall, mentioning HTDP as worthy of studying. Inspired by Mark's question, this paper reports on student performance on Rainfall from an HTDP perspective. Our participants made fewer errors than in prior Rainfall studies and used a diverse set of high-level composition structures; one high-level structure correlated strongly with use of higher-order and built-in functions. Our current hypothesis is that some combination of

[6] Email communication with Ebrahimi, April 2014.

[7] Except for OCaml, the languages used in this study all return exact answers to division, regardless of input type.

using lightweight data structures (in our case, lists), avoiding I/O, teaching data-oriented iterators, and emphasizing a single, widely reusable design method (as in HTDP) underlie these results.

We are not attempting to launch another grenade in the CS1 language-paradigm wars. The ideas in our hypothesis, while hallmarks of functional programming, are hardly unique to it: even Java 8 recently announced the addition of `lambda`, which simplifies writing and using data-oriented iterators. That said, traditional linguistic choices and constraints may well have made Rainfall harder than it needs to be. If students internalize "traverse arrays with `for` loops" and "only create an array if you know the size", they inevitably end up having to interleave multiple tasks in one loop to implement Rainfall. Code that interleaves concerns is arguably harder to implement, harder to test, and ultimately more error-prone than code which keeps them separate. Ginat *et al.* [4] document novices' difficulties interleaving algorithmic plans (though we disagree with their preference for interleaved solutions in CS1).

Whether students learn to separate concerns is a likely a function of how we teach them to program. We suspect that including I/O discourages novices from separating concerns, especially when students are taught to think about programming top-down. Getting input is usually the first step to be executed in a computation. If Spohrer and Soloway's initial-plan model is correct, that input must occur first could well bias students towards picking an input plan first, then adapting it to accommodate other tasks. In contrast, a bottom-up approach that starts with a more behavioral task (such as "find the sentinel") should guide students to first write focused individual functions for other tasks. The influence of I/O on plan selection remains an interesting open question.

This study leaves us thinking about the design principles that students should learn from CS1. We see many students taking CS1 to learn to write small scripts for use in other subjects. They want to grab (noisy) data from the web or write small apps that work on various devices with different input mechanisms. For this environment, students arguably should internalize principles such as "clean your data first", and "separate I/O from core processing". The "Clean First" students in this study landed in this space, but we don't know why. The time appears right to articulate the principles, not just the basic concepts, that our students need, and to reopen studies of planning and plan composition with these principles in mind.

This study leaves us curious about which plans students carry into new languages. We strongly suspect that our same participants would produce solutions with very different structures after learning conventional loops and I/O in CS2 (in Java for most of our participants). New constructs obviously should change how students work, but not at the cost of using solid design principles.

Including students' testing behavior also stands to contribute significantly to plan-composition studies. Test cases can reveal sources of errors: did a student misunderstand the problem, overlook a possible input, or incorrectly implement a correct model of the problem? Prior studies correlated Rainfall performance with language or code comprehension [1, 12]; this makes sense for control-flow errors (i.e., misplaced statements), but less for problem-comprehension ones (i.e., handling division by zero). Testing is lightweight and thus easy to require (as HTDP does) in many functional languages. Indeed, the vast majority of our participants included test cases. We should study whether test cases influence which tasks (and their nuances) students handle.

Going forward, we need data from HTDP students that explores what drives them towards particular high-level structures. We are also extending our project to include other classic CS1 benchmarks (e.g., McCracken). One could argue that Rainfall is biased in favor of functional programming because average has a well-known algorithm (divide sum by count), and both sum and count are standard, heavily-exercised problems in functional CS1 courses. Other classic problems whose algorithms are less evident should prove useful in further understanding the impact of functional programming and HTDP on learning design in CS1.

Acknowledgements: Mark Guzdial encouraged this study during a research visit in Fall 2013. The instructors who contributed data made it happen on short notice. Shriram Krishnamurthi and Matthias Felleisen engaged in helpful discussion of the results. NSF funds supported this work.

8. REFERENCES

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

[2] M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi. *How to Design Programs*. MIT Press, 2001.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[4] D. Ginat, E. Menashe, and A. Taya. Novice difficulties with interleaved pattern composition. In *International Conference on Informatics in Schools: Situation, Evolution, and Perspectives*, ISSEP'13, pages 57–67, 2013.

[5] M. Guzdial. A challenge to computing education research: Make measurable progress. `https://computinged.wordpress.com/2010/08/16/a-challenge-to-computing-education-research-make-measurable-progress/`, Aug. 2010. Accessed April 18, 2014.

[6] L. A. Miller. Natural language programming: Styles, strategies, and contrasts. *IBM Systems Journal*, 20(2):184–215, June 1981.

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

[8] T. Ostrand and E. Weyuker. Collecting and categorizing software error data in an industrial environment. *Journal of Science and Software*, 4:289–300, 1984.

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

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

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

[12] A. Venables, G. Tan, and R. Lister. A closer look at tracing, explaining and code writing skills in the novice programmer. In *Computing Education Research Workshop (ICER)*, pages 117–128, 2009.

[13] P. Wadler. Listlessness is better than laziness: Lazy evaluation and garbage collection at compile-time. In *ACM Symposium on LISP and Functional Programming*, pages 45–52, 1984.