Plan Composition Using Higher-Order Functions

Elijah Rivera
eerivera@brown.edu
Brown University
Providence, RI, USA

Shriram Krishnamurthi
shriram@brown.edu
Brown University
Providence, RI, USA

Robert Goldstone
rgoldsto@indiana.edu
Indiana University
Bloomington, IN, USA

ABSTRACT

Background and Context. Program planning has been a long-standing and important problem in computing education. Finding useful primitives for planning and assessing whether students are able to understand and use these primitives remain open problems. We make progress on this problem by using higher-order functions (HOFs) as planning operations. Not only are HOFs increasingly prevalent in computing broadly, some data science programming sources also recommend their use in planning solutions to data-processing pipelines, giving our task additional applicability.

Objectives. We first wish to confirm that students can understand the behavior of individual HOFs. We also seek to understand which behavioral features of these HOFs are reflected in their understanding. We then investigate their ability to plan solutions as the composition of HOFs. We examine this from two perspectives: recognizing compositions from examples, and describing compositions to solve problems.

Method. Our work is situated in early-stage tertiary education. To investigate student understanding of *individual* HOFs, we present them with both input-output examples and open-ended questions. To investigate the *composition* of HOFs, we use two formats. First, we use the same input-output format as for individual HOFs. We then ask students to construct concrete plans for programming problems defined using text. These plans were constructed using a tool that was custom-built for this work. Students were then asked to write programs to solve the same problems. We perform both quantitative and qualitative analysis of the responses.

Findings. We find students are proficient at recognizing individual HOFs through input-output examples. They use a variety of features to identify HOFs, with the most prominent features being type-based. While they do have difficulty recognizing compositions of HOFs presented in the same input-output example format, there may be simple explanations for this. Either way, students are able to produce correct plans that require composing HOFs, and can successfully translate these plans into correct code.

Implications. We believe work in this vein has many useful consequences for programming education. Explicit planning atop a vocabulary of primitive operations has not received much attention in computing; our work suggests that HOFs provide a useful initial set of planning primitives. (These are, however, generic and not problem-specific, so there is much room for growth.) However, students' possible difficulties in recognizing HOF compositions from examples warrants further attention.

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.
ICER 2022, August 7–11, 2022, Lugano and Virtual Event, Switzerland
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9194-8/22/08…$15.00
https://doi.org/10.1145/3501385.3543965

CCS CONCEPTS

• **Applied computing → Education**.

KEYWORDS

program plans, composition, higher-order functions

ACM Reference Format:
Elijah Rivera, Shriram Krishnamurthi, and Robert Goldstone. 2022. Plan Composition Using Higher-Order Functions. In *Proceedings of the 2022 ACM Conference on International Computing Education Research V.1 (ICER 2022), August 7–11, 2022, Lugano and Virtual Event, Switzerland.* ACM, New York, NY, USA, 21 pages. https://doi.org/10.1145/3501385.3543965

1 INTRODUCTION

Programming is a complex activity that requires significant attention to detail. These details can vary greatly in levels of abstraction, from the very low (e.g., primitive datatype coercions) to the very high (e.g., algorithmic and heuristic choices). Maintaining all this detail can be onerous, with significant amounts of extraneous cognitive load.

For these and other reasons, there is a long history of trying to teach students to *plan* solutions. In principle, plans can focus on high-level solution strategies and avoid some low-level implementation details. By abstracting solutions to these strategies, it should also become easier to identify similarities between solutions and perhaps also transfer knowledge from one problem to another.

Unfortunately, the literature on planning and plan composition has not made much progress over the decades. Studies that began with the Rainfall problem [34] found students *unable* to solve the problem, and the focus shifted to student difficulties rather than student plans. Only in recent years have we seen students successfully able to solve this problem [10, 33] and other problems like it [1, 12].

These recent successes largely ask students to write *programs* and retroactively study the structures that students used. In contrast, we explicitly return to the roots of this problem by asking students to *plan* solutions up front. In particular, we give them a toolkit of planning primitives and ask them to *compose* these into solution structures.

What primitives can we provide? As a starting point, we choose to use built-in *higher-order functions* (HOFs). There is nothing canonical about this choice—one could just as well choose a different origin. We nevertheless choose them for several reasons:

- While they are programming language constructs, they also represent a useful *vocabulary* for data manipulation. As recent work [23] shows, students can understand them from a *behavioral* perspective, not just as patterns of code.
- Influential data science educators like Hadley Wickham encourage their use for data processing [39, 40].
- They are increasingly built into most programming languages (including most recently Java 8). Thus, if students acquire facility with them, that knowledge may be portable (up to the limits of transfer).

Of course, our work is implicitly an experiment: to see whether HOFs work for this purpose. For the style of problems we analyze, they seem to do the job well. This may not generalize to other problems (section 11), and we surely still need a much richer vocabulary of planning primitives (section 13.2). Still, they provide a convenient, concrete, operational starting point.

In this work we first establish students' understanding of primitive HOFs. Then we examine their comfort with HOF composition and with planning. Finally, we examine the transition from plans to code. Concretely, we ask the following **research questions**:

(1) How well do students understand individual HOFs? Concretely:
  (a) To what extent can students recognize example uses of individual HOFs?
  (b) What behavioral features of the individual HOFs do students focus on when identifying these examples?
(2) To what extent can students recognize examples of compositions of HOFs?
(3) How well are students able to plan out a solution to a problem that requires a composition of HOFs?
(4) How well do students do at implementing these plans in code?

2 THEORETICAL BASIS

Our work is fundamentally founded on the theory that students are capable of forming meaningful abstract conceptualizations of HOFs and using these abstractions together in composition to solve problems. This belief is based in prior work from both cognitive science and computing education research. Prior work in *abstraction formation* in cognitive science shows that the presentation of multiple, perceptually disparate instances of the same abstraction (which in our case would be a particular HOF) enables learners to focus on and understand its deep common core [16]. Our recognition instrument (section 6.1) is designed around this principle, using multiple distinct examples to present a single HOF not only to assess student recognition but to aid students in this abstraction formation process.

Our work also builds extensively upon the work done by Krishnamurthi and Fisler [23] (subsequently referred to as KF21). Their work suggests that HOFs can be treated as behavioral building blocks, not only as (say) abstractions over code. Their perspective is essential for our work: it would not make sense to use templates-of-code as primitives for planning in a pre-programming phase. Thus, our work critically depends on students being able to view HOFs as atomic units of high-level behavior.

This in turn means that we are not focused on *schema formation* [30, 38]. While these theories may apply, we believe abstraction formation to be more relevant. Schema formation would primarily apply if we wanted students to recognize the templates-of-code underpinning each HOFs and to be able to use or adapt that template in other similar scenarios, effectively creating their own HOF. Instead, we go the opposite direction, hoping to treat each HOF as a black-box abstraction which can then be used in planning.

We also hold (and in fact this work seeks to test) the core belief that planning is useful for solving computational problems. Specifically, we look at the planning process known in cognitive science as *subgoal decomposition*. In this process, a problem is broken down into small goals, whose solutions are composed together to solve the original problem [6]. For example, the goal of driving one's daughter to school is broken down into subgoals (e.g. getting daughter and self into car, and driving car to school), each of which is broken down into further subgoals (e.g. backing out of driveway, driving on Main Street for 2 miles, taking a right on Elm). Catrambone [4] demonstrated that when learners organize their problem-solving process by subgoal, they are more successful in solving the problem.

Computing problems which involve compositions of HOFs naturally lend themselves to this subgoal decomposition model, and the act of planning is then the act of decomposing a problem into constituent function calls, and then chaining these calls together to create a final program which matches the original specification. Critically, in our work all of this planning is done before any implementation, which forces students to focus on the act of creating and organizing subgoals, and noticing similarities and differences in this organization between problems, all before they have to interact with the syntax of a specific programming language.

3 OTHER RELATED WORK

The field of *plan composition* was originally started by Soloway's study of the Rainfall Problem [34], and has been revitalized recently by progress made by Fisler [10]. The work in this space seeks to understand the process by which students generate code. All of these works attempt to infer a plan or a planning strategy from student-produced code [31], either during [11, 36] or after [8, 21] the coding process. We are not aware of any prior work in the planning space which explicitly discusses planning as a separate step undertaken *before* the act of programming.

Some recent work in plan composition has included information about student *use* of HOFs [1, 10, 12, 33]. We are, however, not aware of literature which directly studies students' *understanding* of compositions of HOFs.

Subgoal decomposition is related to *subgoal labeling*. In subgoal labeling, labels are assigned to different parts of a worked example, which describe the purpose of that individual part to the learner. This process and its benefit to learners in certain contexts has been studied extensively, perhaps most notably by Morrison et al. [26, 27] and Margulieux et al. [24, 25]. However, all of the previous subgoal labeling work has been at a very low level, working with individual syntactic elements of an imperative programming language. None of this work addresses constructs like higher-order functions, which have a more rich semantic behavioral interpretation, or the functional programming space, where the subgoals are not necessarily arranged linearly as they might be in imperative programming. Our work does **not** involve explicit subgoal labeling, but one could potentially view our planning activities as conceptually asking students to come up with their own (unlabeled) subgoals for different parts of the problem.

In a similar vein, one may find commonalities between our work and work done by Muller et al. [29] on *pattern-oriented instruction* (POI), where students are taught to attach labels to specific programming patterns, and to look for these patterns when approaching new problems. De Raadt built on this idea in his dissertation on explicitly teaching programming strategies [7]. But as with subgoal labeling, this previous work has all been at very low level and only in imperative programming languages. Additionally, this work still does not explicitly describe any concrete student plans, and must resort to inferring a plan or strategy from submitted code and/or prose descriptions of the submitted code submitted after the programming was completed.

Another line of work to which we could be compared is the Prolog Tutor [19]. This work builds upon the idea established by Gegg-Harrison [14] that most programs in a domain (e.g., logic programming) can be identified and classified by shared high-level "schemata", or even compositions of these schemata. There is a clear analogy between the Prolog schemata and our HOF abstractions, but like plan composition, this is a line of work in which students are not asked to plan with these schemata *before* programming. All of the systems in this line of work use the schemata to provide feedback to a student once that student has already started engaging with the programming language.

Our work also builds on and goes beyond KF21 in several ways. First, we confirm (section 7.1) their basic experimental results. (In the process, we also extend their results to recognition of compositions (section 8).) Second, whereas they *provide* students with a behavioral understanding (which is not subsequently tested), we *check* whether students have generated one for themselves (section 7.2). The rest of this paper (on composition) is entirely new relative to their work.

4 SCOPE

In this work we limit ourselves to discussions of HOFs that operate on lists. We focus specifically on `map`, `filter`, `andmap`, `ormap`, `sort`, and `take-while`. Except for `take-while`, all of these HOFs exist in most common programming languages, though sometimes by different names. We use Racket's [13] names, following our study setting. The documentation (both definition and examples) for `take-while` were created in the same style as the documentation for the other HOFs in Racket, and was presented to students at the beginning of our first instrument (section 6.1). Table 1 contains a brief description of each of the allowed HOFs. All operations are *functional*, i.e., they create new outputs rather than modify the input.

All of these HOFs have the form `HOF(f, L)` where `L` (the *listarg*) is a list and `f` (the *funarg*) is a function that consumes elements from `L` as input. (Applying these ideas to other datatypes is interesting future work but outside the scope of this paper.)

When we refer to HOF composition, we specifically mean the nesting of different HOF calls to solve a particular problem. The inner HOF may be called in either the funarg or listarg of the outer HOF, and may be nested an arbitrary number of levels deep.

We intentionally left `fold` and its variants out of this study. `fold` is a *universal* function among the HOFs, that is, all of the other HOFs can be written as instances of `fold`. As KF21 report, this makes student responses much more ambiguous and hence difficult to interpret. We do still see some number of students referencing or using `fold` in their responses, even in places where we made effort to exclude it for the purposes of our study, and we view understanding student conceptions of this universal function as important follow-up in this line of work.

5 SETTING

This work is situated at a highly selective private university (tertiary) in the United States. All studies were within the context of an accelerated introductory CS course, which is primarily taken by incoming first-year students. About two-thirds were first-year students (typically 18 years old); the rest had already had at least a semester of tertiary study. About 10% had no prior computing experience, with the rest having taken some (secondary school) computer science, up to the AP CS A course, with a handful having gone farther. Nearly all were new to functional programming.

In order to place into the course, students had to satisfactorily complete a month-long module (over Summer 2021) which teaches beginning functional programming using *How to Design Programs* [9] (HTDP), with graded exercises approximately every ten days. The material of this module roughly compares to the first month of a conventional introductory course at the same university. The module is taught entirely in Racket. The studies are specifically set within this placement process.

The book uses types but as comments (rather than statically checked). This includes parametric types. Every function presented includes its corresponding type signature, including the functions students are asked to write in the module exercises.

The last graded exercise in the module asks students to redo the same problems from the previous exercise, but using only the provided HOFs and no explicit recursion. All of the problems require a composition of HOFs.

This exercise came with a required pre-reading (Part III of HTDP), in which students were first introduced to HOFs. Our first instrument (section 6.1) was included at the end of the reading as a "check your understanding" activity. This activity was not mandatory and not graded, and there were 56 students who completed the activity.

Students were asked to plan out their solutions to this exercise before coding, using a custom variant of Snap*!* [18] (section 6.3). Submitting Snap*!* plans was considered a mandatory part of the exercise, but the plans were not graded. 109 students submitted code for this exercise. Of these students, only 105 students submitted Snap*!* plans. Four of the students who submitted plans did not follow the given instructions, and so we exclude them from analysis to allow for a consistent coding scheme. One student did not submit plans for the first two problems, but we do analyze their plans for the last two problems. One student dropped the class between planning and finalizing their code submission, and so we

Table 1: Higher-order Function Descriptions

HOF	Description
`map`	Transforms the input list point-wise by applying the funarg.
`filter`	Retains those elements of the input list that satisfy the funarg predicate.
`andmap`	Determines whether *all* elements satisfy the provided funarg.
`ormap`	Determines whether *any* elements satisfy the provided funarg.
`sort`	Sorts the input list using the funarg as the ordering comparator.
`take-while`	Retains *all* elements of the input list that satisfy the funarg predicate until the first to fail it.

did not analyze their code, which left some functions unfinished and others unimplemented. One more student submitted code with unimplemented functions, but stayed in the class. We include the functions they did implement in our analysis. In total we have: 101 students who submitted 402 analyzable plans, and of those we have 100 students who submitted 398 analyzable code implementations.

Another instrument (section 6.2) was sent out in the weeks between the end of the placement course and the start of the semester. This instrument was optional and was not considered a part of the required module. 55 students completed this instrument.

We recognize that we may have been able to get more detailed data with closer monitoring of individual students, by assessing their detailed recognition or planning processes. In the pursuit of an inclusive classroom environment for all, we intentionally chose to *not* implement any such system. Surveillance systems can produce significant anxiety in the students being watched [17], which may have altered either the process they used or even their overall performance. Students are particularly likely to feel this anxiety, and show resulting performance decrements, when they perceive themselves to be at risk of conforming to negative stereotypes about their social group [35]. By avoiding imposing these anxieties, we are in fact able to get a more accurate and reliable reading than we might otherwise.

6 INSTRUMENTS

In this section we present an overview of the instruments used in this study. Some instruments may consist of multiple parts that help to answer more than one research question.

6.1 Classifying Examples by HOFs

Before we ask students to plan using HOFs, we want to ensure that students have a grasp of the HOFs we are asking them to use. We use the following instrument to confirm that students are capable of recognizing uses of HOFs. This instrument is inspired by, but significantly extends, KF21, which first studied student understanding of HOFs.

We provide students with 25 questions. Each question consists of a set of three input-output examples (written in Racket syntax), and asks them to identify the (a) single HOF(s) or (b) composition of HOFs that could be used to produce the examples, or to (c) indicate that it is impossible with the HOFs given. Students could only choose from the following five HOFs: `map`, `filter`, `take-while`, `ormap`, and `sort`.

Table 2 shows the breakdown of the tasks. Each single HOF was given in three forms (e.g., Filter1, Filter2, and Filter3) with a consistent structure. We illustrate this through an example first. (The full set of questions is available in appendix A.) Here are the three pairs for Filter1:

```
(list 2 3 6)           → (list 2 3)
(list 5 2 3 6 2 7)     → (list 5 2 3 2)
(list 0 3 9 6 9 1 4) → (list 0 3 1 4)
```

All three can be solved by using the *same* funarg (e.g., one that removes all elements ≥ 6). Here is Filter2:

```
(list "property" "logic" "testing" "code")
→ (list "property" "logic" "code")
(list "friend" "boyfriend" "girlfriend" "weekend" "coffee")
→ (list "coffee")
(list "many" "plethora" "few" "dearth" "solitary" "one")
→ (list "many" "few" "one")
```

Here, each one (can) require(s) a different predicate as a funarg, but they are all of the same type (`String -> Boolean`). In case this problem seems rather hard, recall that students are not being asked to guess what the predicates are; rather, they are *only* being asked to identify which of the five given HOFs could produce this output. Therefore, they only need to know whether this is a "`map` kind of transformation", "`filter` kind of transformation", and so on.

Finally, Filter3 goes even further, varying even the type signature of the funarg:

```
(list "lion" "bear" "tiger")
→ (list "lion" "tiger")
(list (list 1 3 4) (list 2 5) (list 3 7 9 8) empty)
→ (list (list 1 3 4) (list 3 7 9 8))
(list true false true true false false)
→ (list true true true)
```

In general, the HOF1 triples are all produced by the same funarg, the HOF2 triples by different funargs but with the same type, and the HOF3 triples by different funargs with different type signatures.

We refer to these questions as the **Single** questions, of which there are fifteen in all: five HOFs presented in three ways each. The instrument also includes:

- **Multi** Two questions that are intentionally ambiguous and could be solved by either `filter` or `take-while`.
- **Identity** Two questions where the output is the same as the input, and hence can be solved by most given HOFs.
- **Comp** Three questions that can only be solved by a composition of HOFs.
- **Impossible** Three questions that cannot be solved by the HOFs given, even in composition.

Table 2: Classification Instrument: Summary of Questions

Question	Correct Answer	Correct HOF(s)	Notes
Map1	SINGLE	`map`	Same funarg
Map2	SINGLE	`map`	Different funarg with same type signature
Map3	SINGLE	`map`	Different funarg with different type signature
Filter1	SINGLE	`filter`	Same funarg
Filter2	SINGLE	`filter`	Different funarg with same type signature
Filter3	SINGLE	`filter`	Different funarg with different type signature
TakeWhile1	SINGLE	`take-while`	Same funarg
TakeWhile2	SINGLE	`take-while`	Different funarg with same type signature
TakeWhile3	SINGLE	`take-while`	Different funarg with different type signature
Ormap1	SINGLE	`ormap`	Same funarg
Ormap2	SINGLE	`ormap`	Different funarg with same type signature
Ormap3	SINGLE	`ormap`	Different funarg with different type signature
Sort1	SINGLE	`sort`	Same funarg
Sort2	SINGLE	`sort`	Different funarg with same type signature
Sort3	SINGLE	`sort`	Different funarg with different type signature
Multi1	SINGLE	`filter`, `take-while`	Different funarg with same type signature
Multi2	SINGLE	`filter`, `take-while`	Different funarg with different type signature
Identity1	SINGLE	`map`, `filter`, `take-while`, `sort`	Identity function, same type signature
Identity2	SINGLE	`map`, `filter`, `take-while`, `sort`	Identity function, different type signature
Comp1	COMP	`map(filter)`	-
Comp2	COMP	`map(take-while)`	-
Comp3	COMP	`map(filter)`, `map(take-while)`	-
Impossible1	NOT POSS	N/A	Computing average
Impossible2	NOT POSS	N/A	Complicated math
Impossible3	NOT POSS	N/A	Duplicate list

In the process of designing the **Comp** problems, we realized that there are major differences between different kinds of compositional problems. Discussing this is outside the scope of this paper; instead, it is the subject of a separate paper [32]. Following the terminology of that paper, the **Comp** problems used for this instrument only involved *structural* composition.

When answering, students could select from the following options:

SINGLE	"I see how to produce each of these examples with the same (single) higher-order function"
COMP	"I only see how to produce each of these examples using a combination of higher-order functions"
NOT POSS	"I don't think these examples are possible to produce with the given higher-order functions"
IDK	"I don't know"

Students that selected SINGLE were further prompted to multi-select *all possible* HOFs they could use from the list above. Those who chose COMP were asked to indicate (text field) how they would combine the available HOFs. Those choosing NOT POSS were given a text field to justify why they thought it was impossible. Students that selected IDK received no further prompting.

These questions were presented in a random order, except for the first question, which was always Map1. We chose to fix the first question to avoid presenting either a **Comp** or **Impossible** question first, which might have confused students' understanding of the task.

Assessment. The authors carefully designed and studied the problems to have the answers indicated in table 2.

The correctness of any of the **Single**, **Multi** and **Identity** questions can all be checked automatically, by checking how many students selected SINGLE and then, of them, how many selected the correct HOF(s) from the multi-select. For these questions, any student who selected COMP or NOT POSS must also be incorrect, regardless of their textual response, since either selection would exclude a single HOF solution ("I *only* see...").

The **Comp** and **Impossible** answers both have textual responses. Therefore, the authors prepared codebooks, shown in tables 3 and 4, to code these. The codebooks were arrived at by two of the authors after three rounds of coding to obtain a Cohen's κ [5] of 1 for the COMP answers and 0.83 for the NOT POSS answers.

6.2 Behavioral Features of HOFs

Closed-set responses have the advantage of being (mostly) automatically codeable. However, they are insensitive to subtle mental model differences that would not change the response category. Following best practices in educational assessment, we combined the closed-set responses in section 6.1 with additional free-response questions, to build an instrument which is both sensitive and systematic. To do this, at the end of the placement process we include an open-ended question asking students to describe *high-level characteristics* (what

Table 3: Classification Instrument: COMP codes

Label	Definition
VALID-HOF-GROUP	Student proposes a solution which contains a group of HOFs which can be composed into a valid solution for all of the examples
VALID-HOF-INVALID-ORDER	VALID-HOF-GROUP + the student explicitly says how to compose the HOFs, but the composition is invalid for one or more examples because the HOFs are composed incorrectly
INVALID	Student proposes a single solution, and the proposed HOFs cannot be composed into a valid solution for all of the examples
DIFF	Student proposes different solutions for different examples
DIFF-INVALID	DIFF + at least one of the proposed solutions is invalid
VAGUE	Student answer is too short or vague to be labeled with any other label

Table 4: Classification Instrument: NOT POSS codes

Label	Definition
DIFF	Student cites the need for different solutions for different examples
TYPE-VALID	Student makes a valid argument based on the type signatures of the available HOFs
TYPE-INVALID	Student makes an invalid argument based on the type signatures of the available HOFs
HOF-PROP	Student makes an argument based on a property of one or more of the available HOFs, where that property is NOT part of the type signature
HOF-PROP-CANON	HOF-PROP + the property is one of the ones included in KF21
ALTERNATIVE	Student cites a different function (which may or may not be a HOF) which would enable a solution to the problem if made available
NO-PATTERN	Student claims either that they don't see a relationship between the input and the output, or claims that such a relationship does not exist
NO-REASON	Student does not actually present any reason (valid or invalid) for the question being impossible
VAGUE	Student answer is too short or vague to be labeled with any other label

Table 5: Behavioral Feature Instrument: Codes

Label	Definition
FT	Funarg type signature
IT	Listarg type (saying "List" is sufficient)
OT	Output type (saying "List" is sufficient)
WHE/I	Which/how many elements of input determine an output element
OET/I	For list outputs, output element type relative to input
OO/I	For list outputs, output order relative to input
OL/I	For list outputs, output length relative to input
IODIFF	Can output list elements be different from input elements?
OTHER	Other property we didn't account for
OPERATE	Operational description of the HOF
SPEC	Hand-wavy "specification" of the HOF
IDK	"I don't know"

KF21 called "behavioral features") of these functions. The full text of this prompt is as follows:

> In the exercise we asked you to do before Placement 4, we had you classify input-output examples representing uses of different higher-order functions (HOFs). Now that you've had a bit more practice using HOFs, we'd like you to reflect on what specific characteristics of each HOF would help identify examples of that function.
>
> Below we have provided text boxes for some of the HOFs you've worked with so far. Fill in each of them with as many characteristics as you can think of for that function.
>
> For example:
> - "map always produces a list"
> - "the output elements of sort may not be in the same order as the input elements"

If you can't think of anything, it's okay to write "I don't know".

Assessment. Because students were allowed to write free-form textual answers, the authors prepared another codebook for the responses. This codebook is shown in table 5. Because each label can be applied independently of any other label, we compute a Cohen κ for each label after three total rounds of coding between two authors, with a minimum κ of 0.744. The codebook was seeded by the table in figure 1 from KF21.

6.3 Planning with Snap*!*

The final exercise in the summer placement module asked students to complete the same problems as solved in the previous exercise, but using compositions of the allowed HOFs (`map`, `filter`, `andmap`, `ormap`, `foldl`, `foldr`) , with explicit recursion forbidden. Given difficulties we had seen students face in previous years, we decided to ask students to *plan* their solutions before committing to code. This required students to have a more forgiving medium for planning, and in particular a medium that was different from the programming language (because students are often reluctant to change work once they have started doing it [28]).

To this end, we created a custom version of Snap*!*. We deleted all the standard blocks, and created custom blocks for each of the allowed HOFs for this exercise. Our custom blocks do not run or produce output in the interface, since we want students to *plan* in Snap*!* and not to *program* in it. Figure 1 shows the resulting Snap*!* palette.

Note that both variants of `fold` are included in the interface, because the interface was used for all of the problems in the exercise, not just the ones we analyze in this work. We still exclude `fold` from our analysis (section 4). `take-while` was introduced specifically for the recognition activity, but was not necessary for the assignment for which students would be planning, and so was excluded from the interface. `sort` was similarly excluded as unnecessary.

Each of the blocks has input slots corresponding to the arguments the HOF takes. `andmap` and `ormap` are Boolean-producing functions, for which Snap*!* offers a special visual distinction, resulting in those two functions having a different shape than the rest. In hindsight, we should *not* have introduced this additional distinction in our custom version of Snap*!*, given that our version Snap*!* does not enforce this distinction in any way. All of the input slots have the same initial shape (the text box), and will change shape to match any other block that is dragged into the slot.

Students were asked to complete and upload a plan using Snap*!* before they proceeded to coding. They were asked to fill in each input slot with either another HOF block, or with text describing what should go in that input (e.g., a description of a funarg). The ability to write free-form text without running into syntactic difficulties was a particular powerful benefit of Snap*!*. Students were explicitly told to not write Racket code in the input slots (though this did not prevent it from happening).

The exercise contained six problems. Two required `fold`, which we are excluding (section 4). The other four problems are briefly described in table 6, with full problem text in appendix B. All four are string manipulation problems, and are designed to have solutions that require nesting a HOF within the funarg of another HOF.

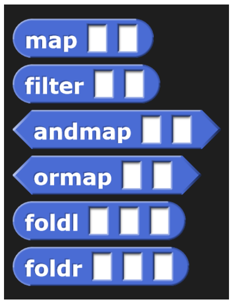


Figure 1: Snap*!* Planning Activity: Palette

Assessment. The nature of the problems means we can divide the analysis into two parts: "do they have the right outer HOF" (OUTER) and "do they have the right inner HOF" (INNER). We also consider the level of detail of the overall plan (COMPLETE).

Because students were explicitly permitted to write their plans partly in prose, we created a codebook to evaluate their responses. To avoid abstracting excessively and losing important detail, we first created a separate codebook for each problem. After three to five rounds for each problem, each of these separate codebooks attained a Cohen κ of 0.821 or greater for each category within the codebook. We then observed that the codebooks were sufficiently similar that we could map them to a single set of codes. To avoid burdening readers with excessive problem-specific detail, we present the combined codes; we believe this does not lose any essential information.

The resulting combined codebook is shown in table 7. In our analysis, we ignore any type-conversions wrappers (e.g., `string->list`) the student may have included, in order to better evaluate the high-level structure of the student plan.

7 UNDERSTANDING INDIVIDUAL HOFS (RQ1)

As discussed in section 1, before we can study whether HOFs are useful primitives for plan composition, we must confirm that students understand and are able to work with HOFs individually. Otherwise we cannot distinguish between failures of general HOF understanding, and failures specifically related to composition.

7.1 Recognizing Individual HOFs (RQ1a)

We confirm student understanding of HOFs using the **Single** questions from section 6.1. A total of 56 students completed this instrument. One of the authors applied the codebook to their responses. The results are summarized in table 8.

We see that students were able to solve most of these with high accuracy (nearly 80%), with a few exceptions worth noting.

Map3 and Sort3 both have lower accuracy, with a high amount of uncertainty (9 and 19 IDK answers respectively). These questions both also had contrived examples with different types and funargs, where at least some of the funargs performed an operation not commonly associated with the respective HOF. Sort2 is contrived to a lesser extent, and we see performance somewhere in between. Based on this, and the otherwise high performance, we suspect that students are simply more proficient at recognizing *natural* uses of

Table 6: Snap*!* Planning Activity: Problem Descriptions

Problem Name	Abbr.	Description
`elim-contains-char`	`ecc`	Given a character and a list of words, produce a list of words excluding words that contain the given character.
`valid-words`	`vw`	Given a list of words and a list of characters, produce a list of only those words that consist of the characters in the provided list.
`l33t`	`l33t`	Given a list of words, produce a list of words where each word has had vowels replaced by a numeric lookalike (e.g., `"example"` becomes `"3x4mpl3"`).
`strip-vowels`	`sv`	Given a list of words, produce a list of words where each word has had its vowels removed.

Table 7: Snap*!* Planning Activity: Codes

Level	Label	Definition
COMPLETE	CORRECT	Plan specifies the full structure of a valid solution, even if some low-level details are incorrect (e.g., improper Boolean negation)
	HOLES	Plan specifies a solution with some pieces left blank which could be filled in to become a valid solution
	IMPOSSIBLE	There is not a readily visible way to transform the structure of the plan into a valid solution
OUTER	CORRECT	Plan uses the same outer HOF as one of the checked common solutions
	OTHER	Plan uses a different HOF from any of the instructor solutions, but one which can plausibly produce a valid solution
	MISSING	Plan appears to be missing an outer HOF and is plausibly correct when placed within a valid outer HOF
	WRONG	Plan uses a different outer HOF from any of the instructor solutions, and from which a solution is not readily visible (this disallows any INNER label)
INNER	CORRECT	Plan explicitly uses the same inner function (`member`, `andmap`, `ormap`, etc.) as one of the checked common solutions
	OTHER	Plan can plausibly produce a valid solution and is not covered by another INNER code
	DECL	Plan declaratively specifies how the inner function is performed, without implementation details, and is plausibly correct
	NO-DET	Plan does not provide details on how the inner function is performed, but depends on it being performed and is plausibly correct
	WRONG	Plan uses a funarg from which a solution is not readily visible

HOFs (a finding that would be in line with related cognitive science literature [22]).

Map1 also showed slightly lower accuracy, but for a different reason. Map1 can only be solved by using a nested call to `build-list` (a HOF that doesn't take in a list but rather a single number) inside of an outer call to `map`. 8 different students cited this fact in their answers, which when added to those who identified the `map` brings the accuracy for this question back up to match the rest.

Table 8 also contains the results for the **Multi** and **Identity** questions. For these questions, most students selected some but not *all* of the correct HOFs, confirming the statement in KF21: students display "[a] failure to appreciate that multiple operators can produce the same results." This result is not too germane to our focus on planning, so we do not investigate this phenomenon further.

7.2 Recognizing Behavioral Features (RQ1b)

55 students completed this activity. All coding was done by one of the authors. The results are shown grouped both by code (table 9) and by HOF (table 10).

With the exception of OET/I, there appears to be broad overall coverage of the features listed by KF21. The input and output types were the most commonly cited, which is also in line with the NOT POSS answers received on the classification instrument, where the most common argument for a problem being impossible was a type-based argument. This suggests that students have been able to form good behavioral models of these HOFs through reading and practice.

We also saw that students did not always decompose their answer into clean-cut properties. 45 students included either *operational* definitions (OPERATE) or "hand-wavy" overall *specification* (SPEC) of some of the functions. For these students, their behavioral conception of the HOFs was presented as a single unit, often

Table 8: Classification Instrument: Single, Multi, and Identity Results

Question	# Correct (/56)	Incorrect	IDK
Map1	41	13	2
Map2	47	6	3
Map3	31	16	9
Filter1	51	3	2
Filter2	47	1	8
Filter3	46	6	4
TakeWhile1	48	6	2
TakeWhile2	44	9	3
TakeWhile3	45	8	3
Ormap1	51	3	2
Ormap2	46	2	8
Ormap3	46	4	6
Sort1	54	1	1
Sort2	42	6	8
Sort3	30	7	19
Multi1	18	33	5
Multi2	17	35	4
Identity1	6	50	0
Identity2	10	46	0

Table 9: Behavioral Features Instrument: Results - Grouped by code

Label	# of students who mentioned at least once
FT	42
IT	54
OT	55
WHE/I	49
OET/I	7
OO/I	48
OL/I	42
IODIFF	35
OTHER	21
OPERATE	35
SPEC	34
IDK	10

Table 10: Behavioral Features Instrument: Results - Grouped by HOF

HOF	FT	IT	OT	WHE/I	OET/I	OO/I	OL/I	IODIFF	OTHER	OPERATE	SPEC	IDK
`map`	10	43	49	42	7	9	34	12	4	32	8	0
`filter`	32	46	52	21	2	19	36	24	5	10	18	0
`take-while`	24	36	43	30	2	17	28	17	12	15	11	9
`sort`	12	40	45	0	0	45	35	21	12	3	12	3
`ormap`	33	45	55	44	–	–	–	–	7	19	18	0
`andmap`	34	45	55	43	–	–	–	–	6	19	17	0

accompanied by the type signature (especially the output type) of the given HOF. However, this could also be an attribute of students not knowing what we were looking for. Had we presented them the rubric (or the table from KF21), they might have done even better. Thus, the above data represent a baseline (for this population and preparation): they get *at least* this far.

At this point we do not go further into analyzing these data. The demonstrated behavioral models of HOFs are sufficient for use when planning, which is the core focus of the rest of this paper. We simply note in passing that even the low score on OET/I is a little misleading, because an understanding of it is implicit in some of their OPERATE or SPEC answers.

7.3 Summary

Students appear to be very capable of recognizing HOFs from examples, and also of coming up with high-level behavioral features. Students often use types to narrow the space of applicable HOFs, even using them to argue impossibility. Furthermore, students are able to do this with just about a week of exposure to HOFs (in this course; some may have had prior exposure), suggesting that the topic—with its potential use as a planning activity—may be amenable to earlier introduction in the curriculum than is traditional. Offsetting this, students seem to struggle with "unnatural" uses of HOFs, at least in this artificial setting (where they have to guess the HOF from just three examples).

8 RECOGNIZING COMPOSITIONS OF HOFS (RQ2)

Satisfied that our students generally understand the individual HOFs, we proceed to understanding how well they were able to recognize these HOFs in composition.

We see lower performance on the **Comp** questions from the classification instrument (table 11). The most common error across all three composition questions was a failure to recognize a need for an outer `map` function, or perhaps a belief that said `map` is simply a formality. The set of three input-output pairs (which all were nested lists, by necessity), were instead treated as a flattened list of nine input-output pairs. If this single error were ignored, accuracy would be comparable to the **Single** questions. What causes this apparent drop in performance: is it cognitive load, something inherent about function composition, the way in which we stated the problem, or something else? This would be interesting to investigate further.

Our code for correct answers to **Comp** questions (VALID-HOF-GROUP) does not require that students specify a particular order when they specify HOFs. We do flag responses that explicitly provide the *wrong* order with VALID-HOF-INVALID-ORDER, but believe that students who had the correct HOFs in the wrong order would immediately realize their mistake in a medium where they could test (or maybe even type-check) programs.

We tested students not only on their ability to classify composition questions, but on their ability to recognize when problems were impossible to solve with composition (table 12).

We see lower performance on the **Impossible** questions than we saw for the **Single** questions, and higher uncertainty as well. Note that we did not ask those who answered IDK to explain their uncertainty, while we did ask those who answered NOT POSS to justify their reasoning. It is possible that some students were less confident in claiming that something was truly impossible, despite having a reasonable justification.

It is worth noting that there was no student who said that any of the **Comp** questions were impossible (NOT POSS), nor was there any student who claimed that the **Impossible** questions were solvable with composition (COMP). Any of the confusions with either set of questions was always either uncertainty (IDK) or a claim that it could be done with a single HOF (SINGLE).

Summary. Once we correct for the "implicit outer `map`" error, we find students are able to recognize composition problems fairly well. It is difficult to tell how to judge their performance on **Impossible** problems: we believe students are rarely given truly impossible problems in secondary school work, so they may simply not believe that NOT POSS is a legitimate answer. Once we also consider the very abstract nature of the task, we feel the students performed quite well.

9 PLANNING WITH HOFS (RQ3)

The heart of this paper addresses *planning* with HOFs as the primitive vocabulary. The earlier sections confirm that students can recognize individual HOFs, understand them from a high-level, behavioral perspective, and can also recognize them in compositions. With this established, we can now meaningfully evaluate the quality of their plans. Concretely, we examine the plans they produced using Snap*!*. We present three representative examples in fig. 2.

We use the three-level structure (COMPLETE, OUTER, and INNER) from section 6.3 and the corresponding codes from table 7 to evaluate student plans. We see how students performed on these three levels in the individual problems in table 13.

We could write at length about these data, but the overarching message seems to be clear: these students, with their background, on these problems, are able to plan very well. The vast majority of their solutions are either correct and concrete, or correct assuming some liberty in writing declaratively using the freedom they were given.

There are a few cases where student solutions are not sufficiently informative (e.g., the NO-DET cases). Further analysis shows that these are not so much indicative of student difficulties as they are of weaknesses of our Snap*!* interface. Concretely, we did not provide a `lambda` block, and students who wanted to describe a procedure were often unclear on what to write in the absence of one. These give us ideas for how to improve our tooling.

Overall, we note that only 10 plans out of 402 were explicitly *wrong*. While of course some "correct" plans could hide misunderstandings, lucky guesses, etc., the overall verdict from this planning activity is a very positive one.

10 TRANSITIONING FROM PLAN TO CODE (RQ4)

The positive outcome of our planning activity can only truly be considered a success if students are also able to translate their correct plans into correct programs. Fortunately for us, this ability was demonstrated in the programs that students submitted. Students are clearly able to implement their correct solutions as correct programs.

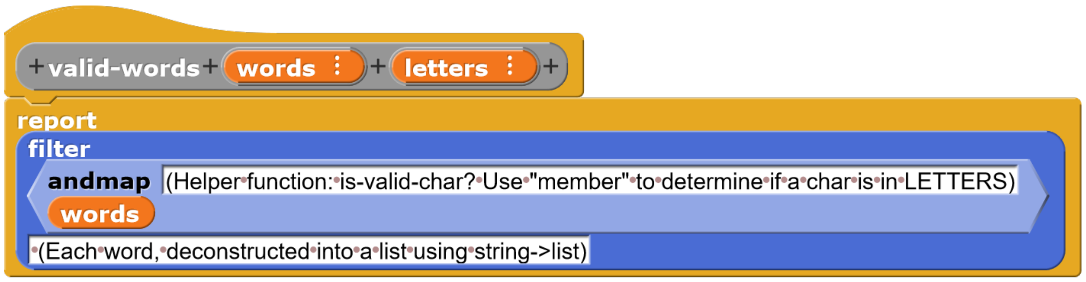
Of the 100 submissions for `elim-contains-char`, 99 submissions for `l33t`, and 99 submissions for `strip-vowels`, **all** of the submissions passed **all** of the instructor-defined test cases. Of the 100 submissions `valid-words`, only seven failed any of the test cases. Two of these seven demonstrated a clear misunderstanding of the problem statement. The students' submitted code and corresponding test cases show that they were checking if all of the characters in the input list of characters are present in a given word, as opposed to checking if all of the characters in a given word are present in the input list: i.e., they swapped the order of the subset relation. The other five failing submissions were all due to a misunderstanding of how `valid-words` should behave in the presence

Table 11: Classification Instrument: Comp Results

Question	# Correct	# Correct but missing `map`	Incorrect	IDK
Comp1	24	20	10	2
Comp2	23	18	10	5
Comp3	32	19	4	1

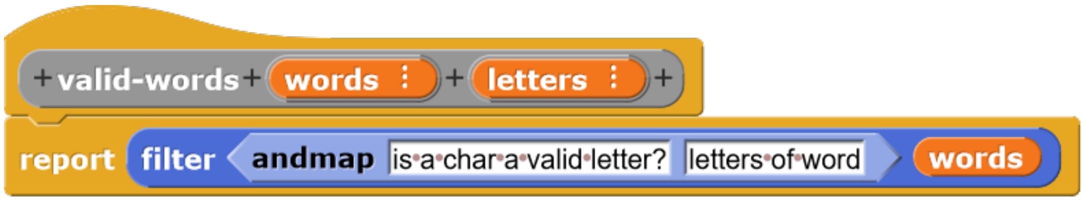
Table 12: Classification Instrument: Impossible Results

Question	NOT POSS (good reason)	NOT POSS (wrong reason)	Incorrect	IDK
Impossible1	33	6	8	9
Impossible2	28	2	2	24
Impossible3	19	7	21	9



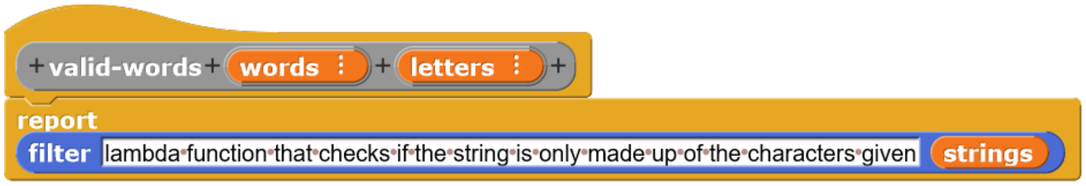
```
+valid-words+ (words) + (letters) +
report
filter
  andmap (Helper function: is-valid-char? Use "member" to determine if a char is in LETTERS)
  words
  (Each word, deconstructed into a list using string->list)
```

(a) This student used nested Snap*!* blocks and very detailed input descriptions.



```
+valid-words+ (words) + (letters) +
report filter andmap is a char a valid letter? letters of word words
```

(b) This student used nested Snap*!* blocks with less detailed input descriptions.



```
+valid-words+ (words) + (letters) +
report
filter lambda function that checks if the string is only made up of the characters given strings
```

(c) This student didn't take advantage of nested Snap*!* blocks.

Figure 2: Three different examples of student plans for `valid-words`

of an "empty" word. The correct behavior was not specified in the problem statement, though it was clarified both in lecture and in a post on the online class forum. Note that the plans that arise from these misunderstandings look nearly identical to plans for correct programs, and the misunderstanding doesn't surface until the implementation step.

Ideally, issues in problem statement understanding are addressed *before* a student attempts to either plan or implement a solution. Thus we don't consider these issues to be a failure of the planning or implementation steps. Ensuring that students understand a problem statement before attempting a solution is tackled extensively in Wrenn's work [41, 42].

Excluding these few failures to understand the problem statement, every other code submission was correct. The overwhelming correctness from both section 9 and these code submissions was excellent to see as instructors, but arguably unfortunate from a research perspective. When designing this research question, we assumed that students might make various mistakes in their plans. If they did, the question would become: would they fixate on these mistakes, or would they be able to recover?

Table 13: Planning Activity: Results

COMPLETE	OUTER	INNER	`ecc`	`vw`	`l33t`	`sv`
CORRECT	CORRECT	CORRECT	39	66	56	70
CORRECT	CORRECT	DECL	49	17	19	10
CORRECT	CORRECT	OTHER	0	0	2	1
CORRECT	OTHER	CORRECT	0	0	1	0
CORRECT	OTHER	DECL	0	0	1	1
HOLES	CORRECT	CORRECT	1	3	5	4
HOLES	CORRECT	NO-DET	6	8	9	9
HOLES	CORRECT	OTHER	1	0	2	0
HOLES	MISSING	CORRECT	0	0	5	5
HOLES	OTHER	CORRECT	0	0	1	1
IMPOSSIBLE	CORRECT	WRONG	3	3	0	0
IMPOSSIBLE	WRONG	-	1	3	0	0

However, the good results from section 9 create a ceiling effect, not giving us room to examine this question to the extent we would prefer. Most students provided correct plans (in varying degrees of completeness), and all students (with few exceptions as noted above) in some way transformed their plan into a correct program.

We do note that even students who submitted plans with errors were able to submit correct solutions. However, we do not have enough insight into their process to identify what enabled them to fix their plan during code construction. At least, it suggests that an incorrect plan is not always an obstacle. We do believe it would be very interesting to examine whether an incorrect *plan* is any more or less of a hindrance than an incorrect (partial) *program* as a starting point.

In principle, a student could start with a correct plan, but run into difficulty with implementation details and end up altering the program to be incorrect. In particular, the transition process is not completely trivial: e.g., students need to insert type-conversion functions like ones to turn strings into lists of characters. We did not see these problems arise with our population of students, and so we choose not to speculate as to what impact these difficulties would have. We instead encourage follow-up studies, potentially with more challenging problems, to see if/when these challenges arise and what impact they have.

11 THREATS TO VALIDITY

We now discuss threats to the validity of our findings. Before going into the details, we note that we view this as a formative study—one of many needed—to make progress on the task of planning.

Internal Validity. We chose problems that were amenable to a two-level HOF composition format, which clearly is not a very general structure. While it enabled us to construct precise codes for evaluating student work, these results may not carry over to other kinds of problem structures. Next, our use of the classification instruments for **Single**, **Multi**, and **Identity** questions is clearly somewhat artificial, and different framings may lead to very different baseline measures (though our findings confirm that at least through this measure, students *do* have a clear understanding of the individual HOFs). Finally, the fact that students were familiar with the problems may have improved their planning ability. All these are factors that can be addressed in follow-up studies, and may affect the findings of this one.

External Validity. Naturally, it is very difficult to generalize from the very particular setting in which we have conducted our study. Everything from our student body to our use of functional programming and the *How to Design Programs* text may be factors. That said, it is quite possible that many of our students are not too different from many students with about a year of tertiary-level computer science. As such, we believe it would be interesting to perform similar studies broadly on students at that stage, across a variety of curricula to identify which factors matter.

Ecological Validity. Some of our activities, like classification, clearly have limited generalizability to the real world. Indeed, they may be sufficiently artificial as to provide only a lower bound on how students would behave when confronted with similar problems in practice. However, we used these activities only for establishing a baseline of student knowledge of the primitive HOFs.

The idea of planning with a tool designed for that purpose is not unreasonable (indeed, various project planning tools, visual designs, etc. are widely used). Furthermore, by embedding the planning activity in a task students were already going to do, we increase its ecological validity. We do not have more insight into their transition from plan to program (section 10) precisely because we did not wish to interfere too much in the way they solved problems during the course, in addition to the reasons described in section 5. (For this reason, students were also not graded on their plans—but took the process seriously and produced quality plans nonetheless!)

12 RESEARCH ETHICS

Per Brown's Institutional Review Board guidelines, our work does not require review. Nevertheless, we have applied standard research protections to protect our students.

13 DISCUSSION

In this work we show that HOFs serve as a viable vocabulary for students to use to plan out solutions to certain kinds of data processing problems. We show that students are able to recognize uses of different HOFs after very little time working with them, and can recognize them both individually and in composition. Students are also able to effectively uses these HOFs to create high-level plans to solve programming problems, and can in-turn create correct implementations from these plans.

We conclude by discussing other lessons learned while doing this study.

13.1 Utility of Snap*!*

As researchers, we found our customized Snap*!* environment to be very useful for both the classroom and our study. Snap*!* provided the right level of abstraction for this study, turning our vocabulary of HOFs into building blocks (so to speak), but leaving space for conveniently writing free-form textual responses. Unlike a pure text solution, Snap*!* provides "structure on the outside" (and indeed as far in as a student chooses to use blocks). This means one can provide reliable, partial, automated feedback: e.g., telling a student that their *plan* is or isn't on the right track before they start to commit to an implementation. One could even imagine assignments where planning is the entire activity, and students are not required to turn the plan into working code (a process that invariably involves a lot of detail that may not be linked to certain learning objectives).

13.2 A Broader Vocabulary for Planning

We believe that as a community, we do not yet have a good enough sense of the *vocabulary* of planning operations that students might employ. We have chosen HOFs because they are reasonably high-level while not being too abstract; they describe a behavioral framework of operation (KF21) while providing freedom (through their funarg) for non-trivial customization. We also hope that learning about planning through HOFs on some problems may transfer to others.

However, HOFs, as with any *generic* library or language construct, are by definition domain-independent. As students plan programs, they presumably use notions from whatever real-world domain they are processing as well, which might employ in a domain-specific vocabulary. Determining how to support these in a planning process that also demonstrates meaningful transfer is an important and wide open problem. What parts are truly still the same in this process, and can be abstracted over? We don't yet have a good enough way to express this abstraction, and while we believe HOFs are a good start, the community would benefit from increasing our vocabulary for planning.

13.3 Forward and Backward Planning

The literature on planning shows that programmers often plan in different directions: from the problem statement "forward", or from the desired solution "backward". Though some studies have ascribed this to different levels of expertise, later research has shown that there is no such sharp line of separation (as Rist [31] summarizes).

Our approach may appear to be agnostic to the direction, but it is not in two subtle ways:

(1) Our students were studying from *How to Design Programs*, which heavily emphasizes a program design "recipe". That recipe drives forward from the structure of the given datatypes, which makes it a forward-planning approach. More recently, other authors [3, 15] have suggested systematic program design methods that drive backward from the desired datatypes. There is currently no analysis of how these two directions compare in a recipe-based program design setting. Whatever design method students have been taught almost certainly impacts the nature of their planning.
(2) Our tool provides a set of planning operations. These operations were based on the input datatype: hence the list-based HOFs (section 4). One could imagine operator palettes that are indexed not only by input type but also by output type. This would enable a more fluid planning process by students—in particular, allowing them to freely switch between the two.

Building the latter form of design tool potentially holds a lot of promise. In particular, when combined with telemetry, it would enable us to study student planning as they are doing it, and gain insight into the directionality of their thinking. Furthermore, we suspect that planning with our blocks promotes a beneficial blend of cognitive loads. The overall cognitive load that makes learning difficult has been decomposed into unavoidable *intrinsic cognitive load* which is related to the inherent complexity of information to be learned, *extrinsic cognitive load* which is imposed by the specific instructional procedures employed, and *germane cognitive load* related to the cognitive resources a learner devotes to the task elements that comprise the intrinsic cognitive load [37]. By isolating planning from coding, Snap*!* reduces the extraneous load associated with superficial programming language details during the planning phase, allowing learners to focus on the combined [20] intrinsic/germane load. Our planning tool could also support an effort to better quantify the cognitive loads [2] associated with planning and coding phases by providing a relatively pure measure of the cognitive load associated exclusively with planning process itself.

ACKNOWLEDGMENTS

We thank the Snap*!* team for creating a very flexible tool, and Jens Mönig in particular for helping us adapt it to this rather unusual purpose. We appreciate the students who worked through these prototypes and ideas. We thank the reviewers and meta-reviewers for their thoughtful comments. This work is partially supported by the US National Science Foundation.

REFERENCES

[1] Peter Achten. 2021. Segments: An alternative rainfall problem. *Journal of Functional Programming* (2021).

[2] J Beckmann. 2010. Taming a beast of burden – on some issues with the conceptualisation and operationalisation of cognitive load. *Learning and Instruction* (2010).

[3] Stephen Bloch. 2010. *Picturing Programs*. College Publications.

[4] Richard Catrambone. 1998. The Subgoal Learning Model: Creating Better Examples So That Students Can Solve Novel Problems. *Journal of Experimental Psychology: General* (1998).

[5] Jacob Cohen. 1960. A Coefficient of Agreement for Nominal Scales. *Educational and Psychological Measurement* 20 (1960), 37–46.

[6] Albert T Corbett and John R Anderson. 1995. Knowledge Decomposition and Subgoal Reification in the ACT Programming Tutor. (1995).

[7] Michael De Raadt. 2008. *Teaching programming strategies explicitly to novice programmers*. Ph. D. Dissertation. University of Southern Queensland.

[8] Alireza Ebrahimi. 1994. Novice programmer errors: Language constructs and plan composition. *International Journal of Human-Computer Studies* (1994).

[9] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2001. *How to Design Programs*. MIT Press. http://www.htdp.org/

[10] Kathi Fisler. 2014. The Recurring Rainfall Problem. In *SIGCSE International Computing Education Research Conference*. 35–42. https://doi.org/10.1145/2632320.2632346

[11] Kathi Fisler and Francisco Enrique Vicente Castro. 2017. Sometimes, Rainfall Accumulates: Talk-Alouds with Novice Functional Programmers. In *SIGCSE International Computing Education Research Conference*.

[12] Kathi Fisler, Shriram Krishnamurthi, and Janet Siegmund. 2016. Modernizing Plan-Composition Studies. In *ACM Technical Symposium on Computer Science Education*.

[13] Matthew Flatt and PLT. 2010. *Reference: Racket*. Technical Report PLT-TR2010-1. PLT Inc. http://racket-lang.org/tr1/.

[14] Timothy S. Gegg-Harrison. 1991. Learning Prolog in a schema-based environment. *Instructional Science* (1991). http://www.jstor.org/stable/23369895

[15] Jeremy Gibbons. 2021. How to design co-programs. *Journal of Functional Programming* 31 (2021). https://doi.org/10.1017/S0956796821000113

[16] Mary L Gick and Keith J Holyoak. 1983. Schema Induction and Analogical Transfer. *Cognitive Psychology* (1983).

[17] Robert L. Goldstone. 2022. Performance, Well-Being, Motivation, and Identity in an Age of Abundant Data: Introduction to the "Well-Measured Life". *Current Directions in Psychological Science* (2022). https://doi.org/10.1177/09637214211053834

[18] Brian Harvey and Jens Mönig. 2010. Bringing "No Ceiling" to Scratch: Can One Language Serve Kids and Computer Scientists?. In *Constructionism*.

[19] Jun Hong. 2004. Guided programming and automated error analysis in an intelligent Prolog tutor. *International Journal of Human-Computer Studies* (2004). https://doi.org/10.1016/j.ijhcs.2004.02.001

[20] S Kalyuga. 2011. Cognitive load theory: How many types of load does it really need? *Educational Psychology Review* (2011).

[21] Philipp Kather, Rodrigo Duran, and Jan Vahrenhold. 2022. Through (Tracking) Their Eyes: Abstraction and Complexity in Program Comprehension. *"ACM Transactions on Computing Education"* (2022). https://doi.org/10.1145/3480171

[22] Kenneth R Koedinger and Mitchell J Nathan. 2004. The Real Story Behind Story Problems: Effects of Representations on Quantitative Reasoning. *The Journal of the Learning Sciences* (2004).

[23] Shriram Krishnamurthi and Kathi Fisler. 2021. Developing Behavioral Concepts of Higher-Order Functions. In *SIGCSE International Computing Education Research Conference*.

[24] Lauren E. Margulieux, Mark Guzdial, and Richard Catrambone. 2012. Subgoal-Labeled Instructional Material Improves Performance and Transfer in Learning to Develop Mobile Applications. In *SIGCSE International Computing Education Research Conference*. https://doi.org/10.1145/2361276.2361291

[25] Lauren E Margulieux, Briana B Morrison, and Adrienne Decker. 2020. Reducing withdrawal and failure rates in introductory programming with subgoal labeled worked examples. *International Journal of STEM Education* (2020).

[26] Briana B. Morrison, Lauren E. Margulieux, Barbara Ericson, and Mark Guzdial. 2016. Subgoals Help Students Solve Parsons Problems. In *ACM Technical Symposium on Computer Science Education*. https://doi.org/10.1145/2839509.2844617

[27] Briana B. Morrison, Lauren E. Margulieux, and Mark Guzdial. 2015. Subgoals, Context, and Worked Examples in Learning Computing Problem Solving. In *SIGCSE International Computing Education Research Conference*. https://doi.org/10.1145/2787622.2787733

[28] Serge Moscovici. 1963. Attitudes and Opinions. *Annual Review of Psychology* (1963).

[29] Orna Muller, David Ginat, and Bruria Haberman. 2007. Pattern-Oriented Instruction and Its Influence on Problem Decomposition and Solution Construction. In *SIGCSE Conference on Innovation and Technology in Computer Science Education*. https://doi.org/10.1145/1268784.1268830

[30] Peter L. Pirolli. 1985. *Problem Solving by Analogy and Skill Acquisition in the Domain of Programming*. Ph. D. Dissertation. Carnegie Mellon University, Department of Cognitive Psychology.

[31] Robert S Rist. 1989. Schema Creation in Programming. *Cognitive Science* (1989).

[32] Elijah Rivera and Shriram Krishnamurthi. 2022 forthcoming. Structural Versus Pipeline Composition of Higher-Order Functions (Experience Report).

[33] Otto Seppälä, Petri Ihantola, Essi Isohanni, Juha Sorva, and Arto Vihavainen. 2015. Do We Know How Difficult the Rainfall Problem is?. In *Koli Calling International Conference on Computing Education Research*. https://doi.org/10.1145/2828959.2828963

[34] W Lewis Johnson-Elliot Soloway. 1984. Intention-Based Diagnosis of Programming Errors. In *National Conference on Artificial Intelligence*.

[35] Steven J Spencer, Claude M Steele, and Diane M Quinn. 1999. Stereotype Threat and Women's Math Performance. *Journal of Experimental Social Psychology* (1999).

[36] James C Spohrer and Elliot Soloway. 1989. Simulating Student Programmers. In *International Joint Conference on Artificial Intelligence*.

[37] J Sweller. 2010. Element interactivity and intrinsic, extraneous, and germane cognitive load. *Educational Psychology Review* (2010).

[38] Jeroen J.G. Van Merriënboer and Fred G.W.C. Paas. 1990. Automation and schema acquisition in learning elementary computer programming: Implications for the design of practice. *Computers in Human Behavior* (1990). https://doi.org/10.1016/0747-5632(90)90023-A

[39] Hadley Wickham. 2014. *Advanced R*. Chapman and Hall/CRC.

[40] Hadley Wickham. 2019. *The Joy of Functional Programming (for Data Science)*. ACM. https://www.youtube.com/watch?v=bzUmK0Y07ck

[41] John Wrenn. 2008. *Executable Examples: Empowering Students to Hone Their Problem Comprehension*. Ph. D. Dissertation. Brown University.

[42] John Wrenn and Shriram Krishnamurthi. 2019. Executable Examples for Programming Problem Comprehension. In *SIGCSE International Computing Education Research Conference*.

A CLASSIFICATION INSTRUMENT QUESTIONS

Map1

```
(list 1 3 6)
 →
(list (list "a") (list "a" "a" "a") (list "a" "a" "a" "a" "a" "a"))

(list 2 1 3)
 →
(list (list "x" "x") (list "x") (list "x" "x" "x"))

(list 4 0 2 1)
 →
(list (list "hi" "hi" "hi" "hi") empty (list "hi" "hi") (list "hi"))
```

Map2

```
(list "Alice" "Bob" "Eve")
 →
(list "Hello Alice" "Hello Bob" "Hello Eve")

(list "eggs" "milk" "cheese" "pan")
 →
(list "eggseggseggs" "milkmilkmilk" "cheesecheesecheese" "panpanpan")

(list "hello" "bonjour" "hola" "aloha")
 →
(list 5 7 4 5)
```

Map3

```
(list (list "lawful" "evil" "rogue")
      (list "neutral" "evil" "mage")
      (list "chaotic" "good" "bard")
      (list "lawful" "good" "paladin"))
 →
(list true true false false)

(list "some" "random" "words")
 →
(list 3 3 3)

(list true false false true true)
 →
(list empty (list "x") (list "x") empty empty)
```

Filter1

```
(list 2 3 6)
 →
(list 2 3)

(list 5 2 3 6 2 7)
 →
(list 5 2 3 2)

(list 0 3 9 6 9 1 4)
 →
(list 0 3 1 4)
```

Filter2

```
(list "property" "logic" "testing" "code")
 →
(list "property" "logic" "code")

(list "friend" "boyfriend" "girlfriend" "weekend" "coffee")
 →
(list "coffee")

(list "many" "plethora" "few" "dearth" "solitary" "one")
 →
(list "many" "few" "one")
```

Filter3

```
(list "lion" "bear" "tiger")
 →
(list "lion" "tiger")

(list (list 1 3 4) (list 2 5) (list 3 7 9 8) empty)
 →
(list (list 1 3 4) (list 3 7 9 8))

(list true false true true false false)
 →
(list true true true)
```

TakeWhile1

```
(list 2 3 6)
 →
(list 2 3)

(list 5 2 3 6 2 7)
 →
(list 5 2 3)

(list 0 3 9 6 9 1 4)
 →
(list 0 3 9)
```

TakeWhile2

```
(list "child" "mother" "father" "uncle" "child")
 →
(list "child" "mother" "father")

(list "duck" "duck" "goose" "duck" "goose")
 →
(list "duck" "duck")

(list "the" "end" "is" "never" "the" "end")
 →
(list "the" "end" "is")
```

TakeWhile3

```
(list 1 7 2 3 -1 -4 2 7 8 -5)
 →
(list 1 7 2 3)
```

```
(list "you" "fool" "you" "absolute" "cabbage")
 →
(list "you")

(list (list true false) (list false) empty (list false) (list true))
 →
(list (list true false) (list false))
```

Ormap1

```
(list 2 3 6)
 →
false

(list 5 2 3 4 1)
 →
true

(list 0 3 1 4)
 →
true
```

Ormap2

```
(list "i" "will" "write" "some" "words")
 →
false

(list "good" "bad" "ugly")
 →
true

(list "correct")
 →
true
```

Ormap3

```
(list 1 3 7 19)
 →
false

(list "rock" "paper" "scissors")
 →
true

(list false false true false)
 →
true
```

Sort1

```
(list 94 26 -3 15 -7)
 →
(list -7 -3 15 26 94)

(list 25 3 25 0)
 →
(list 0 3 25 25)

(list -1 4 -7 0 21)
```

```
 →
(list -7 -1 0 4 21)
```

Sort2

```
(list "epsilon" "gamma" "beta" "alpha" "beta")
 →
(list "alpha" "beta" "beta" "gamma" "epsilon")

(list "highway" "qualify" "subject")
 →
(list "subject" "highway" "qualify")

(list "stool" "counter" "stool" "window" "fan" "window")
 →
(list "fan" "stool" "stool" "window" "window" "counter")
```

Sort3

```
(list 1 2 3 4 5 4 3 2 1)
 →
(list 1 1 3 3 5 2 2 4 4)

(list "lizard" "bird" "dog" "cat" "snake")
 →
(list "dog" "bird" "cat" "lizard" "snake")

(list true false false true true)
 →
(list false false true true true)
```

Multiple1

```
(list 17 84 20 56)
 →
(list 17 84)

(list 94 72 26 -8 -2 -63)
 →
(list 94 72 26)

(list 3 6 13 27 153 252)
 →
(list 3 6 13 27)
```

Multiple2

```
(list "rock" "paper" "scissors")
 →
(list "rock")

(list (list 2 3) (list 1) (list 4 5 2) empty (list 2 7))
 →
(list (list 2 3) (list 1) (list 4 5 2))

(list 1 7 3 -1 -4)
 →
(list 1 7 3)
```

Identity1

```
(list 7 3 0)
```

```
 →
(list 7 3 0)

(list 1 2 1 4 9)
 →
(list 1 2 1 4 9)

(list 8 8 8 8)
 →
(list 8 8 8 8)
```

Identity2

```
(list 4 1)
 →
(list 4 1)

(list (list 7 4 2) (list 5) empty (list 9 3))
 →
(list (list 7 4 2) (list 5) empty (list 9 3))

(list "me" "myself" "i")
 →
(list "me" "myself" "i")
```

Comp1

```
(list (list 4 1) (list 2 7) (list 3 4 9))
 →
(list (list 1) (list 2 7) (list 3 9))

(list (list 7 4) (list 4))
 →
(list (list 7) empty)

(list (list 5 0 3) (list 1 8 4) (list 4 2) (list 3 3))
 →
(list (list 5 0 3) (list 1 8) (list 2) (list 3 3))
```

Comp2

```
(list (list "do" "you" "know" "how" "to" "do" "this")
      (list "i" "know" "a" "shortcut" "i" "like")
      (list "you" "must" "know" "another"))
 →
(list (list "do" "you") (list "i") (list "you" "must"))

(list (list "have" "you" "met" "my" "pet")
      (list "everyone" "i" "have" "met" "has" "met" "him")
      (list "he" "likes" "treats"))
 →
(list (list "have" "you") (list "everyone" "i" "have") (list "he" "likes" "treats"))

(list (list "water" "water" "everywhere")
      (list "our" "body" "is" "mostly" "water")
      (list "well" "water" "is" "from" "a" "well"))
 →
(list empty (list "our" "body" "is" "mostly") (list "well"))
```

Comp3

```
(list (list 1 7 3 -1 -4) (list 3 6 2) (list 4 2 -5))
 →
(list (list 1 7 3) (list 3 6 2) (list 4 2))

(list (list true true false) (list true false false) (list false))
 →
(list (list true true) (list true) empty)

(list (list "green" "blue") (list "red") (list "green" "yellow" "red"))
 →
(list (list "green" "blue") empty (list "green" "yellow"))
```

Impossible1

```
(list 4 3 1 2 5)
 →
3

(list 20 10 15)
 →
15

(list 15 7 8)
 →
10
```

Impossible2

```
(list 1 -8 5 7 -1 -6)
 →
10.488

(list -13 11 16 9)
 →
25

(list -5 0 0 4 -6 18)
 →
24.839
```

Impossible3

```
(list 8 4 6)
 →
(list 8 4 6 8 4 6)

(list "boots" "cats")
 →
(list "boots" "cats" "boots" "cats")

(list (list "x") (list "y") (list "z"))
 →
(list (list "x") (list "y") (list "z") (list "x") (list "y") (list "z"))
```

B PLANNING QUESTIONS

Define the following functions.

- `elim-contains-char :: Char List-of-strings -> List-of-strings`

  Consumes a list of strings and produces a list of the same words, in the same order, excluding those strings that contain the given character.

- `valid-words :: List-of-strings List-of-chars -> List-of-strings`

  Consumes a list of words (represented as strings) and a list of letters (represented as characters). Produces the list of same words, in the same order, that contain only those letters. (Imagine a game where you have to assemble words with just the letters you have.)

  For this assignment, ignore multiplicity: i.e., a word may contain more instances of a letter than the number of times the letter was present in the second parameter. Also, letters should be case-sensitive (e.g., `#\A` does not appear in "cat").

- `l33t :: List-of-strings -> List-of-strings`

  Consumes a list of words (represented as strings) and produces a list of the same words, in the same order, but where some of the letters have been replaced by characters that stand for numbers. Specifically, it turns `#\A` and `#\a` into `#\4`, `#\E` and `#\e` into `#\3`, `#\I` and `#\i` into `#\1`, and `#\O` and `#\O` into `#\0`.

  Note that `#\4` is a character, whereas 4 is a number. You can't do arithmetic on the former or put the latter in a string.

  Note: The first letter of the function name we are asking for is the letter 'l' (Lima), not the number '1' (One). It's pronounced "leet" and is favored in basements worldwide. If you spell it incorrectly, the grading software will not give you any credit!

- `strip-vowels :: List-of-strings -> List-of-strings`

  Consumes a list of words (represented as strings) and produces the same list of words, except with each vowel (`#\a`, `#\e`, `#\i`, `#\o`, or `#\u`, or their upper-case equivalents) removed. (If a word consists of all vowels, it reduces to the empty string but is not removed entirely.)

  Note: Be careful. There's a simple, clean decomposition of tasks in this problem, but if you rush you may end up with a mess.