Student

Identification number	crs2-std1
(Self-reported) **Intended major**	Computer Science
(Self-reported) **Programming languages used before the course**	Java, C++, C#, Python
(Self-reported) **Programming experience prior to the course** (High school class, Online courses (e.g. Coursera), AP class, Programming clubs, Programming boot camps/workshops, Self-study, None, Others)	Self-study, Others: Prior classes from a previous school
Self-assessment of current course performance • A: I can understand the topics very well and have an easy time working on the course assignments • B: I can understand the topics well enough and find the course assignments a bit challenging • C: I find the topics fairly challenging to understand and find the course assignments fairly challenging • D: I have a difficult time understanding the topics and find the course assignments very challenging	A
Consecutive sessions?	Yes

Tasks and references legend

Rainfall tasks • HALT – halt the computation at the -999 sentinel • SKIP – skip negative values (usually with another traversal computation) • TRUNC – produce an intermediate list of values before the -999 sentinel • RNEGS – produce an intermediate list of values without negative values • SUM – produce the sum of a list of values • COUNT – produce the count of a list of values • AVG – produce the avg of a list of values • EMP – handle cases of empty input (empty list, -999 as first element, all negatives)	**Max-Temps tasks** • MXLST – find the max of a list of numbers • MAX2 – find the max between two numbers • BOUT – build the output list of maximum values • RSHP – reshape the input into a list-of-lists
References • [# *] – transcript numbers • {# *} – line numbers in code submission • (FN# *) – field note item number • (S) – scratch work	

Rainfall

Guide Questions	Observations
A. How did they start their process? **1.** How did they interleave their thinking of tasks and patterns? • What patterns did they retrieve first? • What tasks did they identify first? • How did they think through the tasks before starting to retrieve code patterns? **2.** What tasks did they implement first and what pattern did they use to implement it? **3.** How did they use the design-recipe in their process? • What was the role of the different DR components in their work? — for example, whether they used them merely for documentation or to help them get an initial understanding of the problem **B.** What were their difficulties during their process? **1.** What dynamic do we observe in the way they think through the tasks and their implementations? — for example, whether they returned to thinking about the tasks after getting stuck in code, or whether they stayed thinking entirely in code once they started implementing a task **2.** What tasks did they get stuck in? **3.** What tasks were already implemented correctly when they got stuck? **4.** What were they trying to do when they got stuck? — e.g. adding division operation to an add (+) expression	1. He identifies several tasks from the problem statement early on — a. HALT: *"So when -999 is encountered it's the end of reading."* [#2] b. AVG: *"Okay produce the average of the non-negative values"* [#2] c. RNEGS: *"So I need to filter the list of numbers to have only the positive numbers"* [#2] 2. He immediately associates a list-abstraction pattern with the expected general behavior of the function: the expected behavior of producing a singular value from processing a list prompts him towards the use of the foldr list-abstraction, even without mention of a concrete task connected to this: a. *"So it's consumes a list (sic) and produces a number so foldr most likely is my option so let's check with [the course notes]"* [#3] 3.1. Because of (2), he initially jumps into implementing rainfall using foldr, but stops upon recognizing the "special case" of the empty input, which prompts him to go back to using a list-template instead. He writes down the list-template for the rainfall function, but does not let go of the idea of using list-abstractions, thinking that he could use list-abstractions to "simplify" his function later on. a. *"Okay so for list of numbers so I need to use foldr […] well no that's not going to work. I have a special case if the list is empty I cannot compute the value. I need to use cond […] let me rewrite it using the regular list template […] and I will see if I can simplify it using a list abstractions (sic) later"* [#4] 3.2. He writes a signature and purpose statement for rainfall, making sure to capture specific details about the problem in the purpose statement – (1) the average is only computed for positive values and (2) -999 is the terminating indicator for the computation. He writes a couple of trivial test-cases, one from the problem example and another the empty input case, and then leaves this in the meantime. a. Specific purpose statement: *"It takes a list of rainfall readings and returns an average of positive readings […] should specify that -999 is […] the indicator of end of reading the data of interest"* [#3], {#2-3} b. Trivial test-cases: *"let me use the list from the example […] [check] empty list first."* [#3], {#4, 7} 4. The HALT task prompts him towards the idea of creating a function to TRUNC. In touching on this idea, he thinks about list-abstractions he could potentially use to implement TRUNC. He decides to write a list-template function – he starts by writing a signature and purpose that describe his specific goals for the TRUNC function, then test-cases that illustrate his expected behavior of the function, and finally the body of the function. The TRUNC function builds a new list of values until either empty or -999 – he realizes the similar roles of the empty and -999 cases. a. Touches on TRUNC while thinking about HALT, describing the TRUNC-function goal in the purpose statement: *"So I need to somehow filter the list when -999 encountered. I need to stop. […] so first thing I need to do is to make a sub list […] a helper function basically it will check every value and if it is -999 it will stop the function and return whatever it is […] takes a list of rainfall readings and returns the readings prior to -999"* [#5-6], {#42} b. Looks for potentially applicable list-abstraction functions to implement TRUNC, deciding to use a list template when he cannot find applicable list-abstractions: *"So can I use any list abstractions. It takes a list and produces a list – I have [sort and map]. Sort doesn't change the number of items in the list. Map on the other hand also*

5. What patterns did they struggle using?

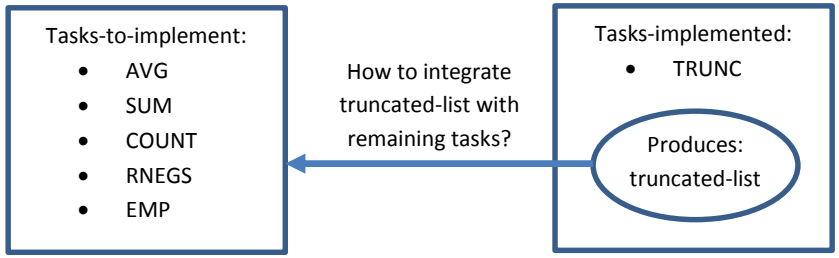
6. How did they try to get unstuck?

doesn't change. Filter. I may use it to filter negative numbers out of the sub list okay, but there are no list abstractions [that can be used for TRUNC]. […] since there are no list abstractions I need to use a regular list template" [#5-7]

c. Recognizes the similarity in roles of the empty and -999 cases (terminating a traversal): *"So if first LON [is equal to -999] what do I do? […] It will just return an empty list right? So I have two cases that return an empty list, I can use probably or statement if it is empty or if it is equal. Yes, I'll combine [the empty and -999 cases]. So it's going to be or empty and then equal, if any of these cases happen, then they return an empty list"* [#8-9], {#50-51}

5. In thinking about his next steps, he simultaneously thinks about several critical components of the problem and how to integrate them – (1) the high-level tasks he needs to achieve (AVG, SUM, COUNT, RNEGS, EMP), and (2) the data produced by the completed parts of his solution (intermediate list from TRUNC). He goes back to his rainfall function and recognizes that the solution he envisions requires deviating from the basic list-template – he adds a local variable definition (before the cond-call) meant to hold a filtered list of values (FILTERED-LIST); within a division-call, writes a call to a sum function using FILTERED-LIST as the argument, and then writes a call to a length function using FILTERED-LIST as the argument. He modifies the empty-case to handle the case when the FILTERED-LIST would return empty by adding a check for a zero-length list.

a. He thinks about how to integrate the data produced by the completed part of his solution with the remaining high-level tasks of the problem: *"I also need to [filter the truncated list] before calling rainfall or at some point in rainfall. […] I need to find the sum of filtered numbers and divide by the list of filtered numbers […] so it's going to divide by the sum of a filtered list"* [#10-13]



b. He uses his task-centric thinking in (5.a.) to guide the structure of his code: *"So I need to step away from the regular list template here. […] I need to create a local constant filtered list so local [define] FILTERED-LIST, so filter LON and that should work, cool. So […] call with sum of FILTERED-LIST constant, divided by length of FILTERED-LIST […] so just add an or with empty […] if LON is empty or [FILTERED-LIST length is 0] return -1"* [#13-15]

c. His code state at this point is [#13-15]:

```
(local [(define FILTERED-LIST (filter lon))]
  (cond [(or (empty? lon) (equals? 0 (length FILTERED-LIST))) -1]
        [(cons? lon) (/ (sum FILTERED-LIST) (length FILTERED-LIST))]))
```

6. After putting the general structure of the tasks into code in (5), he hones in on the specific implementations of each. He recalls the foldr list-abstraction and replaces the sum component with a foldr implementation. Next, he creates an RNEGS-function for removing negatives from the truncated list produced by the TRUNC-function – he writes the signature and purpose to describe the RNEGS-function and decides to use the filter list-abstraction in the

	function body, for which he explains needing another function to use as a predicate within the filter-call. He writes the signature, purpose, and function body for a "non-negative?" function that checks if a value is greater than or equal to zero – he integrates this as the filter-call predicate in his RNEGS-function, using the result of the call to the TRUNC-function as the list argument. He then realizes that the empty-list and other empty-input cases (-999 as first element and all-negatives list) are all similar cases that should return -1, so he modifies his original check to just check if FILTERED-LIST is empty. a. He replaces the sum component with a foldr implementation: *"Instead of using sum, I think foldr is the best solution here. So just foldr […] base case is 0 and the list of numbers is this. So that should return a sum."* [#15], {#13} b. He writes an RNEGS function, which integrates with the TRUNC-function he wrote in (4): *"I need to write a function, filter-list […] So filter-list takes a list of number and returns a list of numbers […] I need to use filter list abstraction […] I need to create another helper function, [so non-negative] takes a number and it returns a boolean so it returns true if the number is non-negative […] because it's a predicate so filter non-negative [and wrap the list of numbers in the sublist function-call]. So this thing returns a sublist of a list of numbers that stops [at -999], next it filters that sublist of non-negative numbers so it should return only [non-negative] readings including 0."* [#16-18], {#17-26, 30-38} c. He realizes that all empty input scenarios can be combined into one case-check: *"If the original list is empty or if the filtered list is empty, wait, instead of writing length equals, I can do the same thing. If empty for filtered list […] if the original list is empty or if a filtered list is empty […] return -1. I don't care if it's an original list, then filter list is also empty."* [#18-19], {#12} 7. To finish his work, he adds test-cases of varying scenarios to his functions – (1) rainfall, (2) filter-list, and (3) non-negative, because from experience, he has lost points for not writing test-cases. He also adds more detail to his purpose statements, for the same reason of losing points in the past for not writing purpose statements. a. One of his motivations for writing test-cases and purpose statements is because of point deduction: *"Let's check if I need any more tests […] I also truly don't want to lose points for not writing check-expects. […] I need to add more wording for [purpose statements] […] I lost a few points in the past for not writing a purpose statement."* [#20-23]; *"the major thing about test cases is not to lose points when grading like when you submit your exercises I've lost up to 15% of my grade for not writing enough check expects"* [#56]

C. What solution-structure did they attempt?

1. What is the solution-structure of their final submission?
2. What components are missing in their final submission?
3. What other solution-structures did they attempt during their process?
4. What prompted shifts from one solution-structure to another?
   - How did students attempt to integrate their next tasks? — e.g. adding a function, adding a parameter, adjusting an expression/construct, etc. (this is code-focused to help identify useful metrics for what students are doing)

Attempted solution-structure style	Clean-first		
Task implemented	**Completed Task? (Yes/Partial/No + Notes)**	**Introduced Type Inconsistency? (Yes/No + Notes)**	**Additional Notes**
TRUNC	Yes	No	[#5-9], {#41-52}
COUNT	Yes	No	[#13-15], {#13} - Used length built-in
SUM	Yes	No	[#10-15], {#13} - Used foldr
AVG	Yes	No	[# 13-15], {# 13}
RNEGS	Yes	No	[#16-18], {#17-26, 30-38} - Integrates with TRUNC - Used filter - Used a separate helper function for the filter predicate
EMP	Yes	No	[#18-19], {#12}

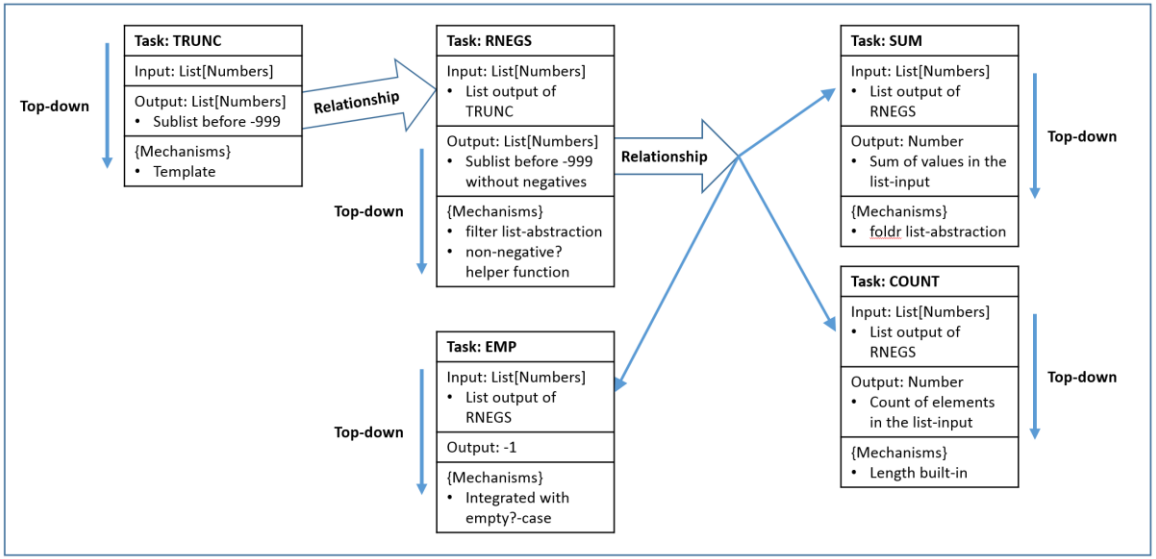
D. Other notes, observations, and summaries

1. The hold of list-abstraction patterns seems very strong, as observed in his early shift towards using them, prompted simply by the idea of producing a singular value from processing a list (A: 2, 3.1, 3.2, 4).
   - Follow-up question: Why wasn't this the case for list-templates – why does the idea of "process a list to produce a value" immediately trigger list-abstractions *instead* of list-templates even when templates are drilled in lists and on other data-structures?
     - Answer: This seems to be an artifact of the instructor's emphasis on grading – in particular, they lose points for not using list-abstractions when they have an opportunity to do so: *"If I don't use list abstractions, I will lose points if there is a point in the program where I could have used but I didn't use them"* [#29]
     - Answer: Another factor that influences his preference for using list-abstractions is the idea that because list-abstractions use less code, there's a lesser chance of committing errors: *"even if I don't lose points, if I use list abstractions, it just makes life so simple and eliminates any possible points where I can make mistakes [because the code is] much shorter"* [#29]

2. He exhibits a top-down programming behavior for familiar components of the problem.
   - The TRUNC task: He starts by looking for an applicable list-abstraction, but when he finds none, he immediately pivots to a template-based function and writes a list-building function that terminates at either empty or -999 – additionally, he recognizes the similar role of empty and -999 as terminators. (A.4)
   - The COUNT task: The idea of counting the elements in a list prompts him towards the length built-in which does this task. (A.5)

- The SUM task: He recalls implementing SUM using foldr and retrieves the pattern for implementation. (A.6)
- The RNEGS task: The idea of removing elements from a list prompts him towards the filter list-abstraction. (A.6)

3. His navigation of the composition of the individual task-components of the problem seems to be directed by a task-level thinking – while he goes through a top-down pattern of implementing each familiar task-component, he describes the composition of tasks through concrete descriptions of the relationships between the outputs of some tasks with the inputs of other tasks.
   - While he works on the low-level (code) implementation of each task, he doesn't lose sight of the high-level task details, which he uses to guide his low-level implementations where needed: He relates how the output of his TRUNC-function is used in the implementation for RNEGS; in turn, he relates how the output of RNEGS can be used for the implementations for SUM, COUNT, and EMP. (A.5-6)
   -



4. **Idea:** The relationship between high-level task-thinking and low-level implementation-thinking seems critical to developing a solution for new programming problems (In the case of Rainfall, "new" could mean a new composition of already-familiar task components).
   - Imbalance – focus on high-level task-thinking: A focus merely on task-thinking, which is possible when there are no low-level patterns to retrieve, means that there are no code-level implementations to write.
   - Imbalance – focus on low-level task-thinking: A focus merely on implementation-thinking, which is possible when students just jump into writing code without a high-level understanding of the individual components of the problem and how they relate to each other, means that composition (or even decomposition) of written code could be difficult because the relationships between the different components (if the components were even identified at all) were not articulated concretely.
   - Balance: Not losing sight of either high- or low-level thinking – high-level task-thinking should guide low-level implementation-thinking, which in turn informs the concrete implementation of high-level tasks.

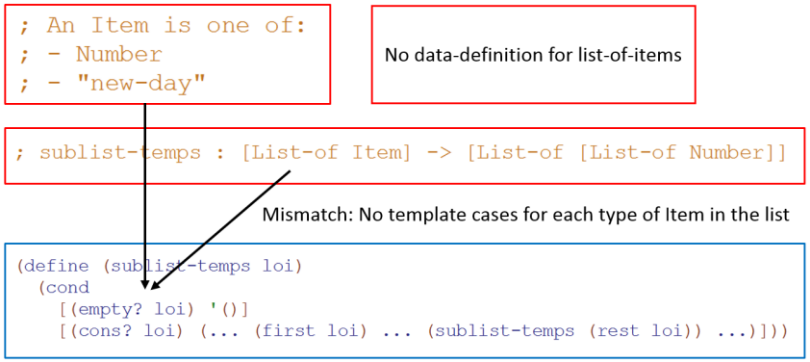
	• Follow-up question: Rainfall has many familiar components. How does this dynamic of high-level task-thinking and low-level implementation thinking look like in more unfamiliar problems like Max-Temps? • Follow-up question: There are students who still fail to write Rainfall, even when many of the task-components should have been familiar to them – why is this the case? Given this idea of the relationship between high-level task-thinking and low-level implementation-thinking, (1) do imbalances in these two levels of thinking manifest among these students, and if they do, (2) how do these imbalances look like, and (3) how do we see these imbalances contributing to students' failures to write Rainfall? • Follow-up question: What "tilts" this balance or what influences the "tilting" of this balance? – Is it the lack of patterns to retrieve, for example, maybe due to new tasks, or new compositions of tasks? 5. His use of design-recipe tools in this problem seems to be mostly mechanical – this could be an artifact of the task components being familiar problems for him, which means that he did not need to rely on the design-recipe to help him write his solution. His explanations about his design-recipe use, however, seem to suggest that he does use them reflectively. • Mechanical use: He writes purpose-statements and more exhaustive test-suites so that he doesn't lose points for not writing them. (A.7) • Reflective use: The signature helps him figure out which pattern to use for implementing a function: *"writing a signature of the function helps a lot actually if you write the name of the function and what it takes and what it produces it sort of helps, oh I have a list of numbers so I might need to use a list abstractions (sic). So then it sort of gives you an idea of where to begin."* [#49] • Reflective use: He writes purpose statements that not only describes the general input and output of a function, but also captures additional details about the problem the function is solving (A.3.2), {#2-3, 18-19}

Max-Temps

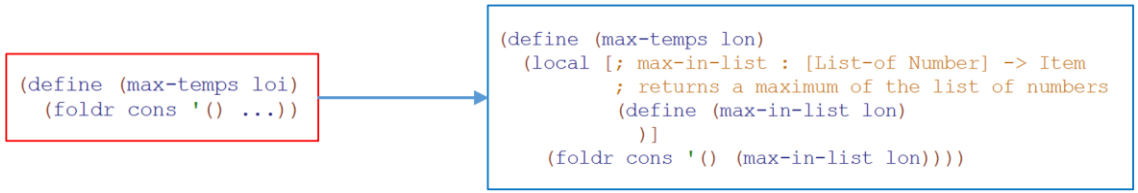
Guide Questions	Observations
A. How did they start their process? **1.** How did they interleave their thinking of tasks and patterns? • What patterns did they retrieve first? • What tasks did they identify first? • How did they think through the tasks before starting to retrieve code patterns? **2.** What tasks did they implement first and what pattern did they use to implement it? **3.** How did they use the design-recipe in their process? • What was the role of the different DR components in their work? — for example, whether they used them merely for documentation or to help them get an initial understanding of the problem	1. He touches on the idea of creating sublists from the main input early on – a. *"okay, takes a list, so it's a list number, number, "new-day" another number "new-day". So I'm going to have to do a sub list out of the list."* [# 2] 2. He starts with design-recipe components – (1) he writes a data-definition that describes the different element-types in the input-list, (2) examples of the elements in the input list (not additional examples of the input-list), and (3) a function template that distinguishes which type an element is. He should have defined another data-definition for the actual list-input using the "Item" definition he wrote, which would be a list-template with cases corresponding to each type of element, in addition to the basic list-cases. a. Data-definition for the input-list element-types – "Item": *"[an item] is one of [number and "new-day"]. So number represents the numeric temperature, "new-day" is a delimiter of the day's temperature readings."* [# 2-3], {# 2-7} b. Examples of Items: [# 3], {# 8-10} c. Function template that distinguishes between Item-types: *"define item-temp I, so it's going to be cond two cases, case one so number? i and then do something with a number. If it is a string? i, do something with the string. I think that's it."* [# 3], {# 11-15} 3. He writes the signature for max-temps using the Item datatype he described in his data-definition in (2). He then checks first for applicable list-abstractions, then defers to the list-template after not finding any, though he doesn't write the template yet at this point, even when he has indicated in the signature he wrote that the input will be a list-type. He then goes through the motions of the design-recipe – defining a purpose statement, examples (empty-list case plus two from the problem statement), and finally, the list-template. a. Signature for the max-temps function: *"So max-temps takes a list of [Item] and returns a list of numbers"* [# 4], {# 19} b. Checks for list-abstractions to use, but defers to the list-template: *"Let's check if I can use a list abstractions (sic) which is most likely a map or filter […] The map applies the same function to the elements so no it doesn't change the number of items […] foldr maybe. Okay. Let's use a regular list [template]."* [# 4] c. Purpose-statement for the max-temps function: *"purpose statement max-temps, accepts a list of Items and returns a list of [max temperatures] for a day."* [# 4], {# 20} d. Examples for the max-temps function: [# 4-5], {# 21-23} e. List-template for the max-temps function: *"define max-temps so list of items LOI […] cond then empty? LOI returns obviously an empty list, if it is cons LOI, deconstruct first LOI rest LOI and call max-temps on LOI."* [# 5], {# 25-26 *m}, (FN# 3) 4. He touches on an algorithm idea for the problem, describing the high-level tasks involved, the order in which the tasks should be carried out, and the output of each task and how they relate to the other tasks – reshape the input into a list-of-lists first, then find the max of each inner-list, storing the maxes generated into a final list-output. a. Describes the high-level tasks (RSHP, MXLST, BOUT) and their relationships: *"I need to take a list and make a list of lists and out of that list of lists, [for every inner-list], find max. […] Yeah I need the list of just lists and for*

	every list on that list of list of temperatures, I will need to find a max and add that max and create a list of all these max." [# 6] 5. He shifts his focus from the overall max-temps function, to a more specific task-component of max-temps that processes an already-reshaped list of lists. In the max-temps function body, he replaces the template code in max-temps and starts writing a foldr list-abstraction, attempting to build the foldr-call with pieces related to the task of building an output list from a list-of-list – (1) he uses cons as the function-argument for building the output-list, (2) he uses an empty-list as the base-case argument, and (3) the input-argument is the output of the call to a helper function that produces the reshaped list-of-lists. The helper function for reshaping the input is undefined at this point. a. Talks about tasks (RSHP, BOUT) and modifies the code towards implementing these tasks: *"I'm not sure if it will work but that's the idea in order to produce a list of list of numbers I need another helper function that takes [that] humongous list and produces a list of list of numbers. Okay, let's try it. Let's leave max-temps for now. Instead of cond, that's just going to be foldr probably, let's do it. So foldr cons empty list and let's just write LON for now. Actually, no just – it's going to be the list that's being created by the [reshaping helper function]."* [# 7-8], {# 25-26}, (FN# 3)
B. What were their difficulties during their process? **1.** What dynamic do we observe in the way they think through the tasks and their implementations? — for example, whether they returned to thinking about the tasks after getting stuck in code, or whether they stayed thinking entirely in code once they started implementing a task **2.** What tasks did they get stuck in? **3.** What tasks were already implemented correctly when they got stuck? **4.** What were they trying to do when they got stuck? — e.g. adding division operation to an add (+) expression **5.** What patterns did they struggle using? **6.** How did they try to get unstuck?	6. Current state of the code at this point: a. Data-definition, examples, and template for Item b. Signature, purpose, examples for max-temps function c. Incomplete max-temps function – contains only an unfinished foldr-call From (5), he again shifts focus, this time towards writing the reshaping function (sublist-temps). He starts with the design-recipe: the signature, purpose, and 3 test-cases (empty-list case plus two from the problem statement) – all of which descriptively illustrate his idea of reshaping the list-input. However, when defining the function body, he writes the full list-template, even though he specifically describes that the input of the function is a list of Items, for which Item could either be a number or a delimiter, as described in (A.2). At this point, it seems that he has lost track of the details of the input and its relationship to the template-structure of the function – each element-type will need to be handled in its own case in the template, thus requiring a change in the basic list-template. At this point, he is stuck figuring out the implementation of the reshaping function and searches for code that can be used to implement the reshaping, but fails to find any. Since the student recognized the underlying list-of-list structure and attempted to approach the problem by reshaping the input, which he couldn't implement in code, he is prompted at this point to switch to an assumption of a list-of-list input. a. Signature for the reshaping function: *"sublist-temps, so takes a list of items and produces a list of list of number"* [# 8], {# 30} b. Purpose-statement for the reshaping function: *"purpose statement, so it takes a list of items and it returns […] a list of lists with numbers"* [# 8], {# 31} c. Examples to illustrate the expected output of the reshaping function: [# 8-9], {# 32-36}

d. Writes a template that does not match the signature and the Item data-definition he wrote: *"So standard template says first item of [list-of-items] rest [list-of-items] and call the sublist-temps recursively in the rest LOI […] I'm thinking how can I modify this template in order […] to take a list of items and produce a list containing other lists with numbers only or an empty list. That's my goal."* [# 9-14], {# 38-41}, (FN# 4)



7. He tries to integrate new code into his existing max-temps function definition (which contains the unfinished foldr-call) – before the foldr-call, he defines a local function (max-in-list) for getting the max from a list of numbers, writing the signature and purpose for it, specifying that max-in-list returns a numeric value (the max of a list). He then wraps the input-argument in the foldr-call with a call to max-in-list, which is incorrect because max-in-list should process the list-elements in the foldr input-argument, not the actual list-of-list input argument.
   a. Signature for local max-in-list: *"[max-in-list] takes a list of numbers […] and returns a max out of these numbers."* [# 20], {# 77}
   b. Purpose for local max-in-list: *"So it returns a maximum […] of the list of numbers"* [# 21], {# 78}
   c. Incorrectly calls the helper on the foldr list-of-list input argument: *"wrap the [foldr list-of-list input argument] in that [max-in-list helper] there we go."* [# 21], {# 84}



8. He starts writing a list-template for max-in-list. He seems to be confused with what to return in the empty-list case due to the special input-case scenario with successive delimiters – based on his examples, a reshaped list-of-lists input will have an empty-list in the place of days without temperature readings. He thinks of an algorithm idea wherein the max-in-list function returns the delimiter if it encounters an empty-day-case (an empty-list), and then once a list of maxes/delimiters is produced, remove the delimiters with the filter list-abstraction. To implement this, he adds a filter call after the local definition, using number? as the function-argument and the result of the foldr-call as the list-input.

a. He decides to handle the empty-day cases by returning the delimiter in the case of empty-lists: *"if it is empty, [max-in-list] shouldn't return any numbers. […] I'm not sure I [should] return anything […] You can probably return an Item and then just filter. […] I can return ["new-day" if the list is empty]"* [# 22-24], {# 69, 81}
b. After getting the list of maxes/delimiters, filter out the delimiters to get just the list of maxes: *"If I have a list of max numbers and ["new-day" delimiters], I then can simply filter that list [and just get the list of maximum numbers]."* [# 24], {# 84}

9. Within the max-in-list function template, he writes cases that correspond to scenarios that the function needs to handle as it traverses the list-of-list input – (1) he writes an empty-case to handle the empty-day (empty-list) elements, (2) he writes a case that returns the last element of a non-empty list to get the max (the terminating case for a non-empty list), which works with the last case of (3) a recursive-case to recursively get the max of a non-empty list. However, he writes the third case incorrectly, recursively calling max-temps instead of max-in-list. He can't figure out his errors at this point and runs out of time. {# 81-83}

```
(define (max-temps lon)
  (local [; max-in-list : [List-of Number] -> Item
          ; returns a maximum of the list of numbers
          (define (max-in-list lon)
            (cond
              [(empty? lon) NEW-DAY]
              [(= 1 (length lon)) (first lon)]
              [(cons? lon) (max (first lon) (max-temps (rest lon)))]))]
  (filter number? (foldr cons '() (max-in-list lon)))))
```

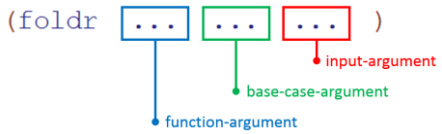
C. What solution-structure did they attempt?

1. What is the solution-structure of their final submission?
2. What components are missing in their final submission?
3. What other solution-structures did they attempt during their process?
4. What prompted shifts from one solution-structure to another?
   - How did students attempt to integrate their next tasks? — e.g. adding a function, adding a parameter, adjusting an expression/construct, etc. (this is code-focused to help identify useful metrics for what students are doing)

Attempted solution-structure style	Reshape-first		
Task implemented	**Completed Task? (Yes/Partial/No + Notes)**	**Introduced Type Inconsistency? (Yes/No + Notes)**	**Additional Notes**
MXLST	Partial - Incorrectly calls max-temps on the rest of the list instead of max-in-list	No	[# 21], {# 84} - Based on his design-goal, should be integrated with the foldr function-argument that processes the foldr list-of-list input-argument
BOUT	Partial - Based on his design-goal, should be integrated with the MXLST function to process the foldr list-of-list input-argument	No	[# 7-8], {# 25-26}
Remove empty-day placeholders from output-list	Yes	No	[# 24], {# 84} - Used filter
RSHP	No	?	[# 8-14], {# 30-41} - Template he wrote in sublist-temps does not match the signature and the Item data-definition he wrote

D. Other notes, observations, and summaries

*For the purpose of discussion, the following terminologies will be used for the foldr function arguments:



Post-mortem

1. **Design-goal:**
   a. His goal was to use foldr to reduce the list-of-lists into just a list of maxes and delimiters (for empty-day cases).

2. **What his design goal requires:**
   a. The input-argument to foldr should be the list-of-lists
   b. The function-argument to foldr should address 2 underlying tasks:
      1. Get the max of each non-empty inner-list in the list-of-lists input-argument (or the delimiter on an empty inner-list)
      2. Append the maxes/delimiters together into one output-list.
   c. The base-case-argument to foldr should be the empty-list, given the goal of producing a list of maxes/delimiters from the list-of-lists input argument.

3. **Primary errors:**
   a. (Relating to E.2.a.) The foldr input-argument he used was the result of calling max-in-list on the list-of-lists – this violates 2 things:
      1. The max-in-list function was designed to work on a regular list of elements, not a list-of-lists
      2. The max-in-list function should be part of the foldr function-argument, because foldr processes the inner-lists from the input-argument one-by-one.
   b. (Relating to E.2.b.) The foldr function-argument he used was just cons, which is only one of the 2 tasks that should be performed (the other being max-in-list) to process the list-of-lists input-argument. He needed to write a function-argument that addresses both tasks.

4. **Other contributing errors:**
   a. In the max-in-list function, instead of recursively calling max-in-list on the rest of the list, he calls max-temps instead.

5. **The combination of the items in E.3 and E.4 above constitute a composition problem** – he had the right pieces of code for the tasks to be implemented, but incorrectly composed them, thus leading to an overall incorrect implementation.

   Hypothesis on the cause of the composition problem – he lost track of 2 critical details:
   a. The input to foldr and
   b. The underlying mechanism of foldr

   Another hypothesis on the cause of the composition problem:
   a. He has not encountered the use of function-arguments that implemented multiple tasks within a list-abstraction, thus, he does not have a pattern to retrieve to implement such a function-argument.

6. **Relationship of the hypotheses to the composition problem observed:**
   a. Given (1) a list-of-lists input-argument and (2) the design-goal of the student, the foldr function-argument should then need to be designed around the dual tasks of (1) finding the max of each inner-list and (2) building the output-list from those maxes/delimiters (as described in E.2.b.)
   b. The foldr function requires keeping track of several things:
      1. The tasks to be carried out to process the input-argument elements
      2. The list-in-process composed of already-processed input-argument elements plus the base-case-argument

3. How to combine the newly-processed element with the list-in-process (which is part of the processing tasks in item E.6.a.)

c. The combination of the factors above could have made the implementation of the design-goal complex for the student. These layers of complexity on the current foldr design-goal may be harder to mentally simulate (another hypothesis on the cause of the composition problem) compared to, say, a foldr using a sum-call over a regular list of numbers, thus leading to the composition problem.

Other analyses

7. He seems to maintain a balance between high-level task-thinking and low-level implementation-thinking, as observed in his structuring of his code based on the task-components he tried to implement. Even with this balance, he runs into a composition problem, which seems to be caused by the underlying complexity of the implementation required by his chosen design-goal.

8. One other point at which he got stuck was writing the reshaping function. As soon as he wrote down the list-template, he simply could not move forward until he was prompted with the assumption of a list-of-lists input to skip the reshaping step. This could be attributed to several factors:
   a. He missed to use the correct template for the input: While he wrote the "Item" data-definition that described the different element-types in the input-list, he did not write the data-definition and template for the actual list-input using the "Item" definition he wrote, which would be a list-template with cases corresponding to each type of list-element.
      1. Follow-up question: Would he have been able to get close to a working reshaping function if he used a template that corresponds to the input-data?
   b. E.8.a. seems to be another instance of him losing track of the details around the input, similar to E.5.a.
      1. Follow-up question: What would be an effective tool that could help students keep track of the inputs/outputs and the relationships of the tasks with the inputs/outputs of other tasks, in addition to the signatures/purpose-statements of the design-recipe?
         - Callback to the task-dependency graphs which made explicit both (1) the tasks that students identified, (2) what the inputs/outputs are, and (3) how the inputs/outputs of the tasks form the relationships between tasks.
         - While the signatures/purpose-statements described details for the function that's being designed, it wasn't designed to make the relationships between functions explicit.
   c. He has not encountered problems that reshaped a list-input with delimiters into a list-of-lists, and thus did not have any pattern close enough to the problem to retrieve. The follow-up question in E.8.a. still holds – if he wrote the right data-definition, would he have been able to at least get close to a reshaping solution even without a pattern to retrieve?

Within-Student Analysis

Guide Questions	Observations
F. Use of the design recipe across the two problems 1. Did they use the design-recipe (or its components) at similar/different times during their process? 2. Did they have similar/different perceptions of the role of the design-recipe (or its components)? – for example, whether they used them merely for documentation or to help them get an initial understanding of the problem	1. He was consistent with his use of certain design-recipe components, particularly the writing of signatures, purpose-statements, and test-cases: Across both problems, he would typically write signatures, purpose-statements, and test-cases before writing his functions. When he wrote purpose-statements, he made sure to capture certain details about the specific function he is describing. He also wrote test-cases with fairly good input coverage, though part of the motivation for this was to avoid "losing points" – he has lost points in course assignments in the past when he failed to write test-suites with ample coverage, as determined by the instructor. Additionally, his data suggests that he uses test-cases to model his design-goal for his functions – for example, even when he cannot implement the RSHP function in the Max-Temps problem, his test-cases are descriptive of his goal for the function. 2. His use of the list-template seems to be overridden by the inclination to use list-abstractions: Across both problems, his first inclination when implementing functions for list-based input is not to use the list-template, but to find applicable list-abstractions for the tasks he identified. **Rainfall:** In Rainfall, this didn't seem to be a problem, as he was easily able to implement SUM and RNEGS through list-abstractions – part of this may be because he has seen patterns for their use in these tasks in lecture, which helped him implement the list-abstraction implementations top-down. Additionally, he also seemed to be able to detect when he needs to deviate from the standard list-template, as shown when he recognized that the body for his rainfall function will need to be changed slightly to accommodate his design-goal of first creating an intermediate list containing only the values of interest. **Max-Temps:** Max-Temps was a similar case, wherein he first looks for potentially applicable list-abstractions before he uses the list-template. He encountered composition problems, however, when he attempted to use foldr to implement BOUT – the composition problem seems to be due to (1) the complexity underlying his design-goal with regard to the use of foldr, as well as (2) the absence of a pattern to retrieve for implementing a foldr with function-arguments requiring the implementation of more than one task. This is elaborated in G.2 below. 3. He wrote data-definitions only in cases when he perceived the input to be unfamiliar to him: He started his Max-Temps process by writing a data-definition for the problem input – this was not something observed in his Rainfall process. When asked about this in the interview, he explains that he did this because the Max-Temps list-input, composed of elements of varying types, was unfamiliar to him. [# 36-38] • This is corroborated by c10's explanation of also writing data-definition only on unfamiliar input

G. High-level task-thinking and low-level implementation-thinking across the two problems 1. Did they move between tasks and implementations in similar/different ways?	1. He showed a balance of high-level and low-level thinking during his process: He moved between high-level task-thinking and low-level implementation-thinking consistently across the two problems. This is evident in the way he talks about the tasks, the tasks' code implementations, and the compositions of his code – he identified the tasks, how the tasks related to each other, honed in on specific implementations of the tasks, and would step back after implementing task code to figure out how to compose the code correctly, by identifying how the output of one task feeds into another task (TRUNC into RNEGS, RNEGS into SUM, COUNT and EMP, for example), for example. He generally had good balance of high-level and low-level thinking during his process. 2. Even with a good balance of high-level and low-level thinking, he ran into composition problems in Max-Temps: Even with this balance between high-level and low-level thinking, however, he failed to write a working solution for Max-Temps. He was able to identify tasks in the problem, and even attempted to decompose his solution into functions or expressions that implemented each task, as evidenced by his attempt to write a separate RSHP function, MXLST function, or use foldr to implement BOUT. He struggled, however, with composing the code for the tasks – his verbal explanations and observations on his process point to several hypotheses for the cause of the composition problems: a. His design-goal may have had an underlying complexity he was not prepared for: This could be seen from his attempt to implement BOUT using foldr – his design goal to use foldr on a list-of-lists to produce a list of maxes required that the function argument handle 2 tasks at the same time: 1. finding the max in a list (the inner-lists within the list-of-list input-argument) and 2. building the output from the maxes. Consequently, this requires keeping track of several things: a. the tasks for processing the input-argument elements, b. the input-argument elements that are already processed, c. the state of the output-list being built, and d. the mechanism for combining a processed-element with the output-list-in-process. Besides (a) and (d), which may be implemented as separate functions (indeed, he implements MXLST albeit with a slight error on the recursive-call), neither (b) or (c) are explicit in the code for foldr – mentally simulating this design-goal for foldr to understand the overall mechanism of foldr requires keeping track of all these 4 things, which may potentially have been too complex for the student. b. He may not have had patterns to retrieve to implement certain task-components: This could be seen in two cases – (1) his attempt to implement BOUT through foldr and (2) his attempt to implement a RSHP function. **For (1):** His code submission, supported by his verbal data, shows that he was able to implement a mostly-correct MXLST function top-down. Indeed, even in Rainfall, most of the task-components were familiar to him, thus he was observed to implement the code for the individual tasks top-down. However, his foldr

	implementation failed because his function-argument did not have the complete set of tasks required to correctly achieve his design-goal – it only had a single cons-call. This isn't surprising, however – exercises on list-abstractions typically used a function-argument that only performed a single task, which is exactly what we see in his code. He has not seen uses of list-abstractions with function-arguments that performed more than a single-task, therefore, he would not have had such a pattern to retrieve to implement his foldr design-goal that required a function-argument that performed more than one task. **For (2):** He was able to touch on the idea of reshaping the input into a list-of-lists. While he was able to touch on this task, reshaping a list with delimiters into a list-of-lists without the delimiters is not a task that he has seen in the course, thus, he had no pattern to retrieve to implement his RSHP design goal, even when he touched on the idea.
H. Course-specific design-practices used across the two problems 1. Did they use similar/different practices?	1. He has habituated practices due to formal instructional rules implemented in the course: • Habit: In both problems, he exhibited a strong inclination towards using list-abstractions to implement tasks for list-based inputs first, which he explains is due to the instructor deducting points from his work when he fails to use them in cases when he could. Additionally, he explains that list-abstractions require writing less code, which in turn reduces the chances of producing errors. This habit does not seem to pose a problem when the tasks he's using them on have retrievable patterns – this is elaborated in F.2 and G.2. • Habit: In both problems, he writes test-suites that have non-trivial coverage, which he explains is again due to the instructor deducting points from his work when he fails to write exhaustive test-suites.