Student

Identification number	neu2018-c10
(Self-reported) **Intended major**	Computer Science and Music Technology
(Self-reported) **Programming languages used before the course**	None
(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)	None
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	C
Consecutive sessions?	No – 3 days in between sessions

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 • DELIM – handle the delimiters (skipping, consecutive delimiters) in some way
References • [# *] – transcript numbers • {# *} – line numbers in code submission • (FN# *) – field note item number • (S) – scratch work	

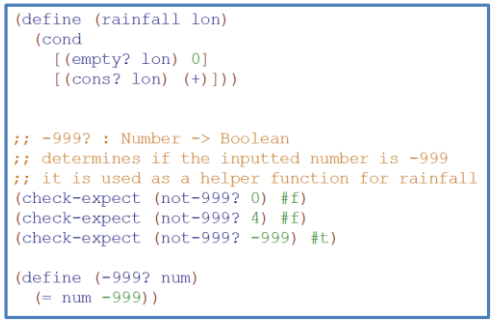
Rainfall

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? 1. 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. She mentions a technique she uses when solving programming problems – she takes note of details from the problem, either highlighting them on the problem-statement or taking notes – a kind of self-regulating habit. She writes a signature for the rainfall function, which is incorrect because both the input and output of the signature notes "list of number" – the output should just be a number (the average of non-negatives); this seems to reflect a misalignment of her understanding with the problem constraints. (FN# 1-2), (S) a. She takes notes and highlights details on the problem-statement as a kind of self-monitoring habit: *"when I read the problem, I also like to underline and at the same time, write it down that we know the function is called Rainfall and then it takes from the list of numbers, and then the list of numbers representing daily rainfall reading. Okay, the list may contain the number -999 indicating the end of the data of interest. So I will circle -999 and then indicating the end of the data."* [# 2], (FN# 1-2), (S) b. She writes an incorrect signature that shows a misalignment between her understanding of the problem and the problem details: *"So it consumes a list of number and also produce list of number"* [# 3], (S) 2. She shifts to the IDE and writes a signature and purpose-statement for the rainfall function – on reading the problem-statement and example again, she realizes that her initial understanding of the output of rainfall is incorrect, and corrects herself, correcting the signature to produce a number for the average instead. She then writes a couple of test-cases, one from the problem-statement and another for the empty-list input, which seems to be habituated since list-problems typically have to handle the empty-list case. Her output for the empty-list case, however, is zero, instead of -1, which further reflects the misalignment of her understanding of the problem still. She begins to write the list-template for the rainfall function, writing the empty-case and then mentioning that she prefers completing the parentheses first to avoid syntax-errors later on – she completes the parentheses for the function without filling in the empty-case; she also does not write the recursive-case of the template yet. a. She writes a signature and purpose-statement for the rainfall function, correcting the signature when she realizes that the output should just be a number: *"I need to follow the data recip (design recipe), so the function name is called Rainfall and then it takes a list of number which represents the daily rainfall and the output is a list of numbers […] So the purpose of the function is that it produces the average of the non-negative values in the list […] up to the number -999. […] I'm wrong on the output. It shouldn't be list of numbers 'cause it's calculating one average number, so it should be just the number"* [# 4-5], {# 1-2} b. She writes a couple of test-cases: *"so we should also check that example and there's a rainfall list 1, -2, 5, -999, 8 and it returns 3. I also want to check-expect for empty list […] it should return a zero"* [# 5], {# 3-4} c. She writes the list-template for the rainfall function up to the empty-case: *"Now I'm going to write the code for the definition of the Rainfall, so define rainfall as this is list of number, I'll just utilize L-O-N that's clear, and then since it takes list of number, so I also need to use cond to also make sure that there's a case for empty list"* [# 6], {# 6-8}

3. In the empty-case of the rainfall function template, she returns zero (again, the misalignment of her understanding of the problem with the problem specifications). In the recursive-case, she partially writes a +-call – from this, she shifts towards the idea of writing a helper-function that checks if a list-value is -999.

   She proceeds to write a "-999?" helper-function: she writes a signature, purpose-statement, and 3 test-cases – all of which illustrate her goals for the function of returning true in the case that a value is -999. She writes the function body for the -999? function. She also explains that she likes to write details about where the helper-function is used in the purpose-statement to inform other people who may read her code about the relationship of the helper function to other functions – she updates her purpose-statement, stating that the function is a helper for the rainfall function. This seems to reflect a value she attributes to the writing of purpose-statements. (FN# 6-9)

   a. She returns a zero in the template empty-case: *"So empty? L-O-N when the list of number is empty, I will just return zero"* [# 7], {# 8}
   b. She writes a partial +-call in the template recursive-case: *"then if the list of number is something, then I will need to plus sums up the —"* [# 7]
   c. She writes a signature, purpose-statement, and test-cases for a -999? helper-function: *"actually I will need to write a helper function that check if the number is the -999. So now I'm going to write the helper function -999?, so that takes a number and then it returns boolean […] This function determines if the input number is -999. […] I need to write check-expect for this and so it's -999? and I will give zero as base case, and it will return the false, and then another check-expect for negative. Okay, maybe I will give one positive number so it make (sic) sure that it works but it also returns to false, then finally, I will have the test that is checking -999 returns true in this function."* [# 7], {# 15-20*}, (FN# 7-9) *This gets replaced in the process and does not make it to the submission file
   d. She explains that she likes to write details about the helper function in the purpose-statement: *"I also always like to write where the function is used. Since I created this function as a helper function, so I will also clearly state that it is used as a helper function for Rainfall. So when other people see it, it's clear that was why I create (sic) this function."* [# 8], {# 17}



```
(define (rainfall lon)
  (cond
    [(empty? lon) 0]
    [(cons? lon) (+)]))


;; -999? : Number -> Boolean
;; determines if the inputted number is -999
;; it is used as a helper function for rainfall
(check-expect (not-999? 0) #f)
(check-expect (not-999? 4) #f)
(check-expect (not-999? -999) #t)

(define (-999? num)
  (= num -999))
```

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?

4. Current state of the code at this point:
   a. The rainfall function is partially written – incorrect output value in the empty-input case and partially written code for sum.
   b. There's a helper function (-999?) that checks if a value is -999.

She attempts to compose her -999? helper function into her rainfall function. In the recursive-case of the rainfall template, she replaces her summing code and adds a cond-call, explaining that this is because there are several cases to be checked when the list is not empty (the summing code gets added back later, see below). She adds a case in the cond-call, and in that case, she writes an if-call – in the test, she calls her -999? function to check if the current list-element is the -999. In the true-case, she returns a zero – presumably, this is to indicate the end of the recursive-sum, which she has written code for earlier (see A.3).

She looks up the Racket documentation to check for the syntax of the if-function. After confirming the syntax of the if-function, she updates the test of the if-call and adds a not-call to her call to -999?. She also updates the true-case of the if-call to do a recursive-sum (completing the summing code she wrote earlier). She realizes that she does not need the additional cond-call within the cons-part of the template and removes that, leaving the if-call in the recursive-case. She misses the fact that she did not yet write code for the false-case of the if-call – she runs her code anyway and it throws back errors. (FN# 10-14)

a. She attempts to compose her -999? helper-function into her rainfall function: *"So now I will need to first check if negative number the first LON is -999?. Actually, I would need Cond in this as well, Cond because even it's the list, it still have many condition, so I will use Cond again and then so it's when the first If, I will write If, 'cause I will need to check if the number, the element of the list is -999, so use the helper function I just created to check if it's -999 of the first and then if it's true, I will just return this to zero […] So now actually, since if it returns true, then we should stop, so I want not so if it's false, then I will not, so if it's not, I will sums up this first LON with the – oh, it's actually a recursive function, so I will also need to do the same -999?. Oh, actually no. I will just do Rainfall and then Rest LON. I probably don't need Cond 'cause I just realized I don't need Cond and then to make it to check if I have enough bracket and it is to see, so If Not-999?, that will take the first LON. If it's true, it takes that value and then go to the first LON and then see how it works by running through the code"* [# 9-11]

```
(define (rainfall lon)
  (cond
    [(empty? lon) 0]
    [(cons? lon) (cond [(if (-999? (first lon))
                            0)])]))
```

→

```
(define (rainfall lon)
  (cond
    [(empty? lon) 0]
    [(cons? lon) (if (not (-999? (first lon)))
                     (+ (first lon) (rainfall (rest lon))))]))
```

5. She modifies her recursive-sum in the true-case – instead of calling rainfall on the rest of the list, she calls the -999? function on the rest of the list. She also adds a false-case, in which she calls the -999? function on the rest of the list. She runs her code, which fails again, and she can't seem to figure out the error – among her errors at this point is the fact that she uses a list "(rest lon)" as an argument to her -999? function, which only takes in a single atomic value. (FN# 15-17)

```
(define (rainfall lon)
  (cond
    [(empty? lon) 0]
    [(cons? lon) (if (not (-999? (first lon)))
                     (+ (first lon) (-999? (rest lon)))
                     (-999? (rest lon)))]))
```

6. She realizes that her call to the -999? helper in the recursive-sum part of her code was incorrect, and changes this back to a recursive-call to rainfall on the rest of the list. She then adds a division operator before the sum call, not realizing the implications of this to the recursive calls – at this point, the state of the code repeats the division, which calls sum, on each recursive-call to rainfall.

   She runs her code, again getting errors. She attributes the errors to the fact that she has not written code for several subtasks – (1) skipping/removing negative list values, and (2) terminating the recursive-sum at the -999. She doesn't seem to recognize that part of her errors stem from the fact that the false-case of her if-call still calls the -999? function on the rest of the list, on top of her misplaced division operation.

   To address the errors, she changes her -999? function to "not-999?", adding a not operation in front of the operation that checks if a value is -999. She also changes the call to -999? in the if-call to "not-999?". She then replaces the false-case of the if-call with a zero, presumably to terminate the function. (FN# 18-23)

   a. She fixes her code for the recursive-sum, but incorrectly adds a division operation before it: *"So [the recursive-call in the summing] should be rainfall as well. I was confused. Sometimes I get confused by just putting the helper function, but let me run through the code again and then see how it goes. Oh, I didn't divide by the number, yeah. I should divide."* [# 16], {# 10}, (FN# 18-20)
   b. She thinks her errors are because she has not written code for several subtasks – skipping/removing negative list-values and terminating the recursive-sum at -999: *"now I have a problem that [the output] is different from 3 because I just noticed that I didn't exclude -2 and also I didn't stop the summation when the -999"* [# 17]
   c. She changes the -999? function to "not-999?": *"let me change the -999? because I want it to be not-999, so I would change my helper function to not-999?, so when the check-expect is zero, it returns a true. When it's a 4, it returns true as well, but when it's -999, it's false."* [# 17], {# 18-23}, (FN# 22)
   d. She replaces the false-case in the if-call with zero, presumably to terminate the function: *"if it's true, it will go to the plus, and then Rainfall. If it's false, it's false, so it will just stop so it returns to zero."* [# 17], {# 12}, (FN# 23)

```
(define (rainfall lon)
  (cond
    [(empty? lon) 0]
    [(cons? lon) (if (not (-999? (first lon)))
                     (/ (+ (first lon) (rainfall (rest lon))))
                     (-999? (rest lon)))]))

(define (-999? num)
  (= num -999))
```

→

```
(define (rainfall lon)
  (cond
    [(empty? lon) 0]
    [(cons? lon) (if (not-999? (first lon))
                     (/ (+ (first lon) (rainfall (rest lon))))
                     0)]))

(define (not-999? num)
  (not (= num -999)))
```

7. In her division-call, she adds a length-call after the sum-call – she calls length on a call to rainfall on the input-list. Syntactically, this completes the formulaic structure of the average formula, though this composition is incorrect, as the current state of the code builds the sum and repeats the count components recursively while also calling

division on each recursive-call – an effective average formula code would only call division once, and on the sum and count components which have already been computed prior to the division.

The incorrectness of this composition is corroborated by the way she talks about the average-computation part of her code. Her discussion of her average code highlights 2 things: (1) her description is primarily formulaic and (2) seems to be tightly bound with the recursive code structure she has already put in place – her discussion centers on the sum and count being built recursively, which lacks the critical insight that the sum and count components need to be completed first (whether separately through different functions or together through accumulative parameters, for example) prior to calling division on the two computations., (FN# 24)

a. She adds a length-call within the division-call to implement average – her discussion of the average computation code is primarily formulaic and tightly bound to the code she has already written: *"I will need to write this function as I will need divide by the number of the number that I add, so it would be of the length of Rainfall. Actually, the length of not the Rainfall of the LON. […] it check (sic) if the first element of the list is not -999, and since it returns to true, it will go through the sum of the first LON and the rainfall rest LON, divide by the length of rainfall LON."* [# 18-19], {# 10-11}

```
(define (rainfall lon)
  (cond
    [(empty? lon) 0]
    [(cons? lon) (if (not-999? (first lon))
                     (/ (+ (first lon) (rainfall (rest lon)))
                        (length (rainfall lon)))
                     0)]))
```

8. In addition to thinking about how to make the code for her average-computation work, she also thinks about how to integrate her current code with the code for other tasks, particularly the task of terminating at the -999. However, she seems to primarily focus on figuring out what code to put in, even without a high-level insight of how the tasks should compose in the first place (cutting the data at the -999 before doing sum and count, for example).

   She replaces the call to rainfall in the length-call with a call to the filter list-abstraction, using the not-999? function as the function argument – while this is her attempt at writing code that terminates the computations at -999, there is no specific explanation as to why she did it this way. She runs out of time before she figures out how to correct his. {# 10-11}, (FN# 25-26)

   a. She uses the filter list-abstraction to terminate the computation at -999 within the length-call: *"Maybe [the function is in an infinite loop], that's why it can't [stop]. It's [in an infinite loop], so it gets the error, so now I want to use length and I use list abstraction which is Map, uhh, filter -999 of LON. […] So the filter, according to the [course notes], so it returns a sub-list of the input list containing just those elements which the function returns true"* [# 19-20], {# 11}

```
(define (rainfall lon)
  (cond
    [(empty? lon) 0]
    [(cons? lon) (if (not-999? (first lon))
                     (/ (+ (first lon) (rainfall (rest lon)))
                        (length (filter not-999? lon)))
                     0)]))
```

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?
   2. 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	Single-traverse		
Task implemented	**Completed Task? (Yes/Partial/No + Notes)**	**Introduced Type Inconsistency? (Yes/No + Notes)**	**Additional Notes**
HALT	Partial	Yes - In her process (B.5), she calls (rest lon) on her -999? helper-function (this eventually becomes "not-999?"), which only takes a singular value, not a list. (FN# 15-17)	[# 19-20], {# 11}
COUNT	Partial - Incorrectly composed within the AVG computation (see AVG)	No	[# 18-19], {# 11}
SUM	Partial - Incorrectly composed within the AVG computation (see AVG)	Yes - In B.5, she calls -999? on the rest of the list (see also HALT)	[# 7, 9-11], {# 10}
AVG	Partial - Retrieved the average formula, but focused on code implementation, failing to step back and analyzing how the high-level tasks composed. This resulted in calling division repeatedly (over each recursion), with a sum component that was still being built recursively instead of being built prior to the division.	No	[# 16], {# 10-11}

RNEGS	Partial	No	
EMP	No	No	

D. Other notes, observations, and summaries

1. She still struggles with syntax in writing her function calls, which suggest that her skill in composing expressions and/or functions is still lower than, or not completely at, the relational-level.

    3. **In B.4:** Even when she has looked at the documentation for the if-syntax, she still makes simple syntax errors by forgetting to add code for the if-function false-case and running her program anyway. While she resolves this later on, she only does so after seeing the error.
    4. **In B.5**: She introduces type-inconsistencies in her function call to her -999? helper-function – the function was designed to have atomic values as input, but she calls the function with list arguments twice, in the recursive part of the sum-call and the false-case of the if-call. She doesn't recognize these errors until later when she runs her code.

```
(define (rainfall lon)
  (cond
    [(empty? lon) 0]
    [(cons? lon) (if (not (-999? (first lon)))
                     (+ (first lon) (-999? (rest lon)))
                     (-999? (rest lon)))]))
```

2. Her process suggests an imbalance between the low-level code implementation and the high-level task-view of the problem.

    5. **In B.7:** Once she retrieved the average formula and focused on the code to implement the formula, she failed to step back to get a higher-level insight on how the task-components of the formula actually work.
        - Another way of looking at this problem is that it is the failure to see the boundaries of the 2 pieces of knowledge – knowledge of the (1) template and (2) the average formula – and how they can be correctly combined to form the correct solution.
    6. **In B.8:** She was able to grasp the task of terminating at the -999, but solely focuses on implementing code which she perceives to address the task (using filter with the not-999? helper-function within the length-call), without stepping back to think about how the task specifically interacts with other tasks in the problem.

3. She still doesn't have a strong grasp of list-abstractions, which led her to incorrectly compose the code for a truncate-task with the count-task.

    - **In B.8:** She misinterprets the concept that filter produces a sublist from a list-input and assumes that composing this with her not-999? helper-function would produce the sublist of items before the -999, which in turn led her to compose this with her length-call, thinking that the expression would produce the count of values before the -999.

4. She explains that her initial understanding of the problem was that rainfall would produce a list of non-negative numbers before -999. This explains her scratch work and (A.1-2) in which she wrote a signature with lists of numbers for both input and output. [# 26-31], (S)

    - Another misalignment of her understanding with the problem specifications that she discovered post hoc was the output value of rainfall when the average cannot be computed – she only realized that the output should be -1, which explains why she returned 0 in the case when the input-list is empty. [# 33-34], {# 8}

5. She finds value in particular design-recipe components, as well as holds malformed understandings of the purpose of other components.

    - She explains that she generally follows the signature-purpose-test-code pattern, explaining that that was the process they were taught in the course.
    - **Value – self-regulating/self-explanation of the details of the function and confidence in her work process:** She attributes values to components of the design-recipe, particularly the writing of the signature and the purpose-statement – these help clarify the input, output, and purpose of the proposed function. Additionally, following the recipe makes her feel more confident with her work. [# 35-46]
    - **Value – default starting point for writing code:** She explains that because she is a beginner with no prior programming experience, she only draws on the knowledge she gained from the course and thus mostly follows the design-recipe when working on her programs. She claims to find the design-recipe helpful when working on bigger codebases because she always goes back to the templates to write her functions. There is no further elaboration on why this practice is valuable to her. [# 64]
    - **Malformed understanding – data-definitions only for new/unfamiliar data:** However, when asked about other recipe components, such as the data-definition and data-examples, she seems to have a malformed understanding of the purpose of these components. She explains that they're taught to only write data-definitions when there are new data in the problem – this explains why she did not write the data-definition for the rainfall input; lists of numbers are typical of problems in the course, which may have led her to feel that she did not need to write data-definitions. [# 35-46]
    - **Follow up question:** Would she have written a more effective solution if she wrote a data-definition and realized that each type of value within the input-list (negative, non-negative, -999) requires a different kind of computation?

6. She perceived the new composition of familiar tasks in Rainfall as challenging.

    - She describes the tasks of excluding the negative values from the computation and terminating the computation at the -999 as confusing. She explains that while she may have encountered similar tasks (average, skipping values, terminate at a value) in previous problems, she has not previously encountered a problem such as rainfall that combines all of the tasks together in a single problem. [# 48-54]

7. Her motivation for the decomposition of code is that it enables testing individual components of the program.

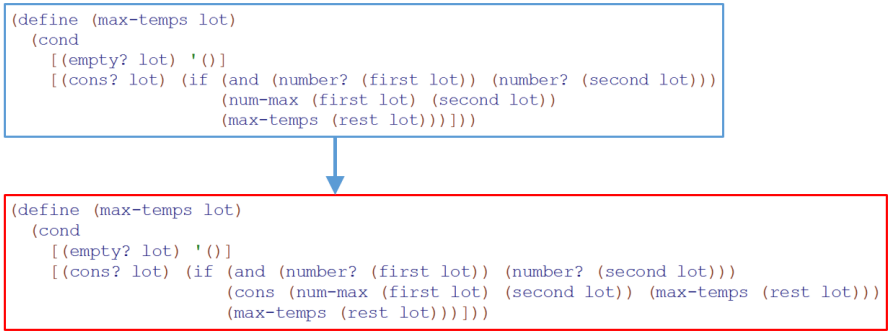
    - When asked about her reasoning on why she pulled the not-999? function into its own function, she explains that doing so enables her to test the functionality of that function separately from the rest of the program.

	This seems to be a value that she attributes to the decomposition of her code – it enables unit testing of individual components of the program. [# 56] • **Follow-up question:** What's curious about this is why she didn't do the same for the sum and count components of the average computation. If she stepped back to reflect on the high-level composition of the tasks instead of focusing on the low-level code implementation of the formula, would she have realized the underlying decomposition of the tasks (sum and count)? 8. She has this understanding of local definitions that defining local functions instead of defining them globally will make a program run faster in terms of time complexity. [# 56-60] 9. She admits that she should have made an effort to understand the problem better before jumping to immediately writing code. [# 70]

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? 7. 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. Just like in Rainfall, she first highlights and sketches out parts of the problem-statement, seemingly as a habit of self-regulation. She writes the max-temps signature on paper, with comments on the details. (FN# 1-2) a. *"I'm thinking I need to write a function that's Max-temps and then that takes list, one list, and that returns a list of numbers, and the Max-temps takes into the list of numbers and string "new-day" and the numbers represent temperature readings as taken by a sensor and the "new-day" string is sent at the start of each calendar day, so whenever the "new-day" occurred, the later number that starts is the on the other day."* [# 3], (FN# 1-2) 2. She proceeds to write the num-max helper function, following the design-recipe steps as she goes along – she writes a signature, a purpose-statement, and 3 test-cases. She identifies 3 different cases to address in the function: num1 > num2, num1 < num2, and num1 = num2 – because of this, she writes a cond-call with a case for each of the cases she identified. She runs her code without errors. {# 8-18}, (FN# 3) a. She writes the signature and purpose-statements for num-max: *"so now I'm going to first write the Num-max that's provided in the text, so Num-max, it takes two numbers, so it's number, number, and then produce a number as well. So the purpose statement of this function is output the bigger number of the between number between the inputted two numbers."* [# 4], {# 8-9}, (FN# 3) b. She identifies 3 cases to write tests on: *"So example is Check-expect Num-max of 0 and 2. That will output 2. Check-expect another case is Num-max there's a negative number and then 4. That will take 4. […] So if the inputted two number has the same value, it will just return that value"* [# 4, 8], {# 11-13}, (FN# 3) c. She writes the num-max function with a cond-case for each of the cases she identified: *"so after the test, I will have to write Define Num-max of Number 1, Number 2, so this is Cond and there will be three conditions, so actually, yeah, it will be If Compare If, actually I will do if first number is equal to second number, then I will output either Number 1 or Number 2. I will just put Number 1, and if Number 1 is larger than Number 2, then it will still output Number 1, but if Number 1 is less than Number 2, which is Number 2 is bigger, then I will take Number 2."* [# 8], {# 14-18}, (FN# 3) 3. She then shifts to defining the max-temps function and again follows through the design- recipe. She starts by writing the signature – she first writes that the input is a list of number and string, which she changes to a list of "Temperature". Because "Temperature" is an undefined data at this point, she writes a data-definition for Temperature, noting that Temperature could either be an empty-list or a list containing a number and a string – this is incorrect because Temperature refers to the elements of the input list, which is either a number or a string. As it stands, she seems to have thought of the data-definition to be describing the input-list itself, which is still incorrect as it should have been either a list of a number and a list-of-temperature, or a list of a string and a list-of-temperature. Writing data-definitions would have been the ideal course-of-action, especially since she has not encountered mixed-datatype list-inputs in the course, but this suggests 2 things – that her knowledge of how to write data-definitions may be fragile, or that even the design-recipe steps may also need to be taught within other input-type contexts.

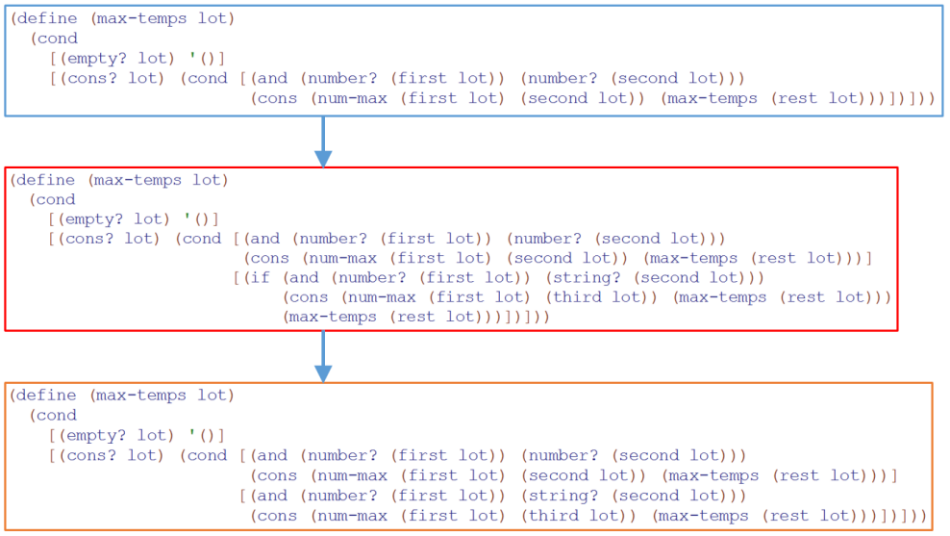
	a. She writes the signature for max-temps with "list of Temperature" as input: *"[Max-temps] takes in a list of number, number, and string. So list of number, actually, I will say list of temperature and then it outputs a list of number"* [# 9], {# 23}, (FN# 4) b. She writes an incorrect data-definition for Temperature: *"so now I need to define the temperature, so temperature is one of either empty list or list of number and string, and temperature where number represents the temperature readings and the string represents is "new-day", which represents the start of each new calendar day."* [# 9], {# 1-5}, (FN# 5) 4. She proceeds through the design-recipe, writing a purpose-statement and test-cases – she writes a test-case for each of the 2 examples given in the problem-statement, adding a test-case for an empty-list input because of the list-input. a. She writes a purpose-statement for max-temps: *"So the Max-temps, it outputs the maximum temperature of each day based on the given list of temperature"* [# 10], {# 24}, (FN# 6) b. She writes a test-case for an empty-input and each of the 2 problem-statement examples: *"So for example, since I already have example from the text, so I will just copy and paste that, which is and then test Check-expect Max-temps of list of 40, 42, "new-day", and 50, and "new-day", 52, and 56, so this will output a list of 42, 50, 56. Okay. Another example that I have provided is similar but it's list of 40, and 42, and there's a consecutive string of "new-day", and 50, and "new-day", and 52, 56. This will output list of 42, 50, and 56. And notice since it takes in the list of temperature, it takes in list data, so I would need to do the Check-expect of Empty List and it will produce an empty list as well as its – yeah, it will return empty."* [# 10-11], {# 25-29}, (FN# 6)
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. Current state of the code at this point: a. She has a working num-max function, with a signature, purpose-statement and 3 test-cases – a case for each of the cond-cases in the function definition). b. She has a signature, purpose-statement, and 3 test cases (2 from the problem-statement and 1 empty-list case) for max-temps. c. She wrote an incorrect data-definition for "Temperature" She immediately jumps to writing code for the max-temps function, even when she hasn't articulated the specific tasks within the problem that she wants to address. She starts writing the list-template and returns empty in the empty-case. Immediately, she starts populating the recursive-case with code that refers to the specific input-examples provided in the problem-statement – she writes an if-call, checking if the first and second list-elements are numbers. In the true-case, she returns the result of calling num-max on the first and second list-elements; in the false-case, she calls max-temps recursively on the rest of the list. At this point, she has introduced several errors: (1) the true-case returns a numeric value, which introduces a type-inconsistency and violates the max-temps signature, and (2) she recursively calls max-temps on the rest-of-list, instead of just the suffix of the list without the first sublist. She then runs her code, which fails. She recognizes the type-inconsistency in the true-case and

5. What patterns did they struggle using? **6.** How did they try to get unstuck?	fixes this by adding a cons-call before the num-max-call, with a recursive-call to max-temps on the rest of the list – this also serves to build the output list. She runs her code, which fails again. Additionally, she describes updating the purpose-statement of num-max to note that the function is used as a helper-function for max-temps. This is similar to her behavior in Rainfall of adding detail in helper-function purpose-statements to describe where the helper-functions are used in. a. She starts writing the list-template, returning empty in the empty-case: *"So for list empty, LOT, then it would just return to empty list"* [# 12], {# 32}, (FN# 6) b. She populates the recursive-case with code that refers to the specific input-examples in the problem-statement: *"if it's Cons? LOT, then I will need to use if the first is number? […] I will just do Number? and do first of LOT, and then I will need to do and (the "and" function) condition because I want Number? of rest of LOT. So if there are number, I will use the Num-max – the helper function that I utilize, that I just made before I start coding. […] and then if the first element and the second element of the list of temperature are the number, then I will use the Num-max function to compare them. […] If the first and second and not the number, then I will just run Max-temps on the Rest LOT. […] okay, so although I run the Num-max but I need to output a list, so I will do cons before Num-max and then I will do the Max-temps of Rest of LOT, and then let's see if it will make the list that I want or not. […] So if the first element of the list of temperature is number and the second element of the list of numbers or elements is number as well, then it creates a list and do Num-max of First LOT and Second LOT, and then it does the same thing for the rest and then if both are false, they do the rest of the LOT."* [# 12-15; *This gets updated during the process and does not make it to the submitted version.], {# 35-36}, (FN# 6)  6. She attempts to diagnose her error and replaces the second-calls with rest, which introduces a new error – she is now calling num-max on a list, rather than a numeric value, which is another type-inconsistency. She runs her code, which reveal precisely this error, and she reverts her code. (FN# 6-8) 7. She modifies her code by moving the entire if-call she wrote into an inner cond-call in the recursive-case of max-temps. She then uses the condition in the if-call as the first case in the inner cond-call, and combines the true- and false-cases from the if-call into the answer for the cond-case.

She then adds a second case in the inner cond – she writes an if-call, checking if the first list-element is a number and the second list-element is a string. In the true-case, she builds a list (with cons) with the result of num-max on the first and third of the list, and a call to max-temps on the rest of the list. In the false-case, she calls max-temps on the rest of the list. She runs her code which fails because (on top of her previous unresolved errors), she did not add an answer to her new cond-case.

She tries to resolve the error by repeating what she did with her first case. She uses the condition in the second if-call as the second case in the inner cond, and combines the true- and false-cases from the if-call into the answer for the second cond-case. She doesn't realize that she was not supposed to compare values from different sublists, which adds to the list of errors her design currently has. (FN# 9)

a. She moves her if-call into an inner cond-call: *"I will need Cond here again […] and then I will just put the thing that I made into the first condition. So the first condition is if two elements, they both are number, then I will compare to and then do the rest"* [# 18], {# 35-36}, (FN# 9)
b. She adds an if-call in a second cond-case to check if the first element is a number and the second element is a string: *"if the first element of the given list is number and second element of the list is string, then I will do cons and then Num-max of First LOT, then the third LOT, and then after using that, I will do Max-temps of the Rest LOT as well, and if both are false, then I will just go to Max-temps of Rest LOT"* [# 18], {# 37-38}, (FN# 9)
c. She repeats what she did on her first inner-cond-case and re-organizes the new if-call as the second inner-cond-case: *"so I just put the first clause of the If as the question clause in the Cond and then I will go to this and now I don't need this, and so something to fix the second Cond […] Yeah, I don't need the second Max-temps of the Rest of the LOT"* [# 19], {# 37-38}, (FN# 9)



```
(define (max-temps lot)
  (cond
    [(empty? lot) '()]
    [(cons? lot) (cond [(and (number? (first lot)) (number? (second lot)))
                        (cons (num-max (first lot) (second lot)) (max-temps (rest lot)))])]))
```

```
(define (max-temps lot)
  (cond
    [(empty? lot) '()]
    [(cons? lot) (cond [(and (number? (first lot)) (number? (second lot)))
                        (cons (num-max (first lot) (second lot)) (max-temps (rest lot)))]
                       [(if (and (number? (first lot)) (string? (second lot)))
                            (cons (num-max (first lot) (third lot)) (max-temps (rest lot)))
                            (max-temps (rest lot)))])]))
```

```
(define (max-temps lot)
  (cond
    [(empty? lot) '()]
    [(cons? lot) (cond [(and (number? (first lot)) (number? (second lot)))
                        (cons (num-max (first lot) (second lot)) (max-temps (rest lot)))]
                       [(and (number? (first lot)) (string? (second lot)))
                        (cons (num-max (first lot) (third lot)) (max-temps (rest lot)))])]))
```

8. She cannot figure out the causes of her errors at this point. To experimentally debug, she defines constants for the list-inputs she uses in her test-cases – unsurprisingly, this does not fix her errors.

She also adds another case in the inner-cond, this time checking if the first list-element is a string, the second is a number, and the third is a number – the answer is a call to cons with a call to num-max on the second and third elements of the list as the first argument, and a recursive-call to max-temps on the rest of the list as the second argument. She attempts to add code in the second inner-cond to check if the third element is a number, and the first inner-cond-case to check if the third element is a string, but runs out of time before finishing her code. At this point, all of the cases in the inner-cond are designed only against the example input in the problem – she does not consider input wherein the inner-lists have arbitrary data.

a. She defines list constants to experimentally debug her code: *"let me try to define a constant to see if it gives the same error or not. So I will define this one as the list of this – LIST1 and then this will be define LIST2, and then I just create the constant to see if this same error will occur"* [# 20], {# 20-21, 26-29}, (FN# 10)
b. She adds another case in the inner-cond: *"if the first element of the list is string and the second element is number of second LOT, and then the third element is number as well, then I will cons of the Num-max of second LOT and third LOT, and then do the same thing to the Max-temps of Rest of LOT."* [# 21], {# 39-40}, (FN# 11)

```
(define (max-temps lot)
  (cond
    [(empty? lot) '()]
    [(cons? lot)
     (cond
       [(and (number? (first lot)) (number? (second lot)) (string?
        (cons (num-max (first lot) (second lot)) (max-temps (rest lot)))]
       [(and (number? (first lot)) (string? (second lot)) (number? (third lot)))
        (cons (num-max (first lot) (third lot)) (max-temps rest lot))]
       [(and (string? (first lot)) (number? (second lot)) (number? (third lot)))
        (cons (num-max (second lot) (third lot)) max-temps (rest lot))])]))
```

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?
   8. 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	Single-traverse: process-until		
Task implemented	**Completed Task? (Yes/Partial/No + Notes)**	**Introduced Type Inconsistency? (Yes/No + Notes)**	**Additional Notes**
BOUT	Partial	Yes - See B.5, she returns the result of num-max in the true-case of the if-call, violating the max-temps signature	[#], {#}
MAX2	Yes	No	[# 4, 8], {# 8-18}, (FN# 3)
DELIM	Partial	No	[# 12-21], {# 35-40}
			[#], {#}

D. Other notes, observations, and summaries

1. **Her design-goal is centered around a single-traversal of the input, which necessitates the following tasks:**

   1.1. Getting the max between 2 values
   1.2. Keeping track of the current max of the current sublist
   1.3. Building the output-list
   1.4. Handling delimiters

2. **Her process can be primarily characterized as a very low-level, code-focused process.**

   a. She recognizes some of the tasks that she needs to address from the problem-statement – finding the max of two values (because of num-max), building an output list, and skipping delimiters (because of the examples) – but she's never concrete about how the tasks related to each other.

      For example, she never talks about how she plans to address the idea of finding the max of successive numeric values (beyond two values) or how to handle the delimiters and how this delimiter-handling relates to building the output-list (see the necessary tasks for her design-goal in E.1.). Even when she was not having any success with runs of her code, she never steps back to think intentionally about what the exact high-level tasks are and how those tasks need to be put together.

   b. From B.5., it can be seen that she jumps directly from the problem-statement to writing code she perceives is relevant.

c. Her code-focused process can also be gleaned from how she just directly wrote code pertaining to the specific cases given in the problem-statement examples. Towards the end, she even loses focus on which pairs of values to apply MAX2 on, comparing values before and after a delimiter.

d. Her explanation of her max-temps function confirms that she was indeed designing the function based only on the first example in the problem (no successive delimiters). She also realizes that the second condition was incorrect (see B.7.b-c) – she realized that instead of comparing numbers separated by the delimiter, she should have just created a list and recursively called max-temps on the rest of the list, presumably to get the max of the succeeding sublist. [# 36-46]

3. **Part of her difficulty may be attributed to the absence of concrete patterns required for the design-goal of her solution** (see E.1.)
   a. She explains that the design-recipe helps her keep track of what she is doing, presumably to help her self-regulate. However, she also explains that she got stuck because she thinks that she has never learned ways to solve problems such as max-temps from the lecture – this seems to confirm the fact that she has not been taught patterns for solving problems such as max-temps. [# 52-55]

   b. Patterns that are necessary to actualize the tasks, but she may not have seen in the course include:
      1. Keeping track of a value by modifying the list-input – in this case, replacing the two elements being compared at the head of the list with the resulting maximum value, and then repeating the process on succeeding numeric elements.
      2. Recurring on a part of the list that is not just the rest of the list – in this case, when encountering a delimiter after getting the max of the first sublist, recur on the part of the list after the delimiter (consecutive delimiters may need additional code to handle such cases).
      - See the relevant maxtemps-corrected Racket file as reference for potential implementations of these patterns.

4. Her description of her experience with learning in the course suggests that she has a difficult time figuring out which patterns (such as list-abstractions, for example) are applicable to which type of problems. [# 67-68]

   a. She explains that her homework partner has been helpful in explaining to her when to use list-abstractions for the problems they work on in the assignments. [# 71]

5. She explains that they are taught to write data-definitions if they "see a new type of data" – in this case, she has not seen list-inputs with elements of varying datatypes, which drove her to write a data-definition for the input. [# 54]
   a. This corroborates her thoughts about writing data-definitions in her Rainfall data.
   b. She wrote an incorrect data-definition for a new datatype, "Temperature" – she seems to refer to "Temperature" as the input-list itself, which requires a different data-definition from what she wrote down. This may suggest 2 things:
      1. Her knowledge of writing data-definitions may be fragmented in some way

	2. Design-recipe components may also need to be taught within other input-type contexts (somewhat similar to the idea of teaching the use of additional accumulators, or using accumulators to accumulate other types of values besides atomic-values, or recurring on a part of the list beyond the rest).

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. She was consistent with her use of certain design-recipe components, particularly the writing of signatures, purpose-statements, and test-cases: Across both problems, she would typically write signatures, purpose-statements, and test-cases before writing her functions. It seems, however, that she is able to write more exhaustive test-cases for tasks she is familiar with, but not the other functions – there seems to be an underlying relationship between task-familiarity and the use of test-cases: if she has seen or implemented a task before, she may already have an idea of the space of input for that specific task and thus writes a more exhaustive coverage of test-cases for that task. This is not something observed with other tasks – in rainfall, she only writes the test-case from the problem-statement plus the empty-input, which is typical of list-based problems, but writes more for her sentinel-identifying helper-function. In max-temps, she only writes the test-cases from the problem-statement, but writes a more exhaustive coverage for the num-max helper-function. This, in turn, also seemed to have influenced the design of her code, writing code only against the limited examples/test-cases she has written. Idea: There is a disconnect between how we envision/want students to use examples/test-cases and how students use them in practice (this is not new). The above concretely illustrates a disconnect between what we want students to do with examples and test-cases and how students use it: ideally, we want them to practice writing examples and test-cases as an advanced planning mechanism to see the space of the inputs and ideally, make them think about the behavior of their code in advance of their implementation. In c10's case, there seems to be 2 different things happening depending on her familiarity of the tasks she's writing examples for. For familiar tasks, she seems to write a fair coverage of examples, presumably because of existing knowledge of the task. For unfamiliar tasks, she defaults to what is provided in the problem-statement, or stereotypical examples, such as empty-list examples for list-based problems. This, in turn, seems to limit how she thinks about her solution, as observed in her max-temps solution where she only designed her code strictly based on the examples from the problem-statement. While the idea of using test-cases/examples to drive code design is a goal of the design-recipe, c10's case seems to be a premature use of it. She wrote conditional-cases within her code that only handles the ones in her test-cases, which does not have coverage of other input scenarios. Her design-goal is thus limited only to the specific examples she's designing with and does not meet the required functionality of the problem (in addition to her other errors). This raises the following question – would she have written more effective code if she had written more examples and test-cases of varying input scenarios? 2. She attributes value to particular design-recipe components: She explains that she likes writing the signature and purpose-statements because these help clarify the input, output, and purpose of proposed functions, which, in turn makes her feel confident about her work. This is also evidenced by the fact that she writes more detailed purpose-statements, particularly for her helper-functions – in addition to describing the proposed function, she adds details explaining where the helper-function is a helper to.

	This is observed in her process across both problems. In Rainfall, she wrote a purpose-statement for the not-999? helper-function relating it to her rainfall function, in Max-Temps she wrote a purpose-statement for the num-max helper-function relating it to her max-temps function. 3. She wrote data-definitions only when she perceives the input to be unfamiliar: When asked why she didn't write a data-definition for the Rainfall input, she explains that they're taught to only write data-definitions when there are new data in the problem – she thus did not write the data-definition for the rainfall-input because lists of numbers are typical of problems in the course. On the other hand, she wrote a data-definition for max-temps because she was unfamiliar with input-lists with elements of varying datatypes. The data-definition she wrote, however, was incorrect (see Max-Temps E.5.b. and A.3.), which may suggest 2 things: a. Her knowledge of writing data-definitions may be fragmented in some way b. Idea: Design-recipe components may also need to be taught within other input-type contexts (somewhat similar to the idea of teaching the use of additional accumulators, or using accumulators to accumulate other types of values besides atomic-values, or recurring on a part of the list beyond the rest).
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. Her movement between tasks and code is greatly skewed in favor of writing code: Her thinking around tasks only goas as far as identifying the tasks, but not the deeper compositions of those tasks. She seems to instead push all of her thinking around composition to her code implementations. She focuses on an identified task, attempts to implement code for the task, and then attempts to add on code for the other tasks that she has identified, without concretely identifying the high-level relationships that exists between those tasks. **Rainfall:** The failed compositions of her code for average reflects her failure to step back and think concretely about the high-level tasks and their compositions. It can be clearly seen that she retrieved the formula pattern for average, as well as concrete patterns for the task-components of average: a recursive-sum and count, but failed to account for how the outputs of those tasks integrated with the division code. In addition to not having thought about the problem at the task-level, the task for terminating at the sentinel seemed to be one for which she also did not have a concrete pattern to retrieve. She attempted to integrate it at several points of her code, but cannot correctly figure out where the sentinel-code should be placed – this absence of a pattern can be seen from her incorrect attempt to use the filter list-abstraction to implement TRUNC (Rainfall E.3.). **Max-Temps:** In Max-Temps, she generally has identified 3 distinct tasks: MAX2, BOUT, and DELIM. Her process, however, shows that she immediately jumped to, and focused on, writing code for these tasks without prior planning or a high-level insight about how the tasks need to be composed – this could be seen in how she wrote code on-the-fly and used trial-and-error to figure out next steps. While she wrote code based on the test-cases, which suggests a process of relating the design of the code to illustrated patterns of behavior, she seems to have done this prematurely (see F.1.). Her design-goal for max-temps required keeping track of the current max of the current sublist as the input is traversed, as well as building the output whenever a delimiter is encountered. Her code attempts touch on

	patterns related to this design-goal (see Max-Temps E.3. for the necessary patterns for this design-goal), but these are not patterns she has seen in the course. This suggests then that while she may have touched on these ideas for her design-goal, she did not have the concrete patterns required to actualize her goal. Idea: Knowledge of patterns may be a binding component between high-level tasks and low-level code. This seems to illustrate a relationship between two of the skills in the SOLO framework: composing expressions and decomposing-tasks. Students generally reach the relational-level for composing-expressions early on in their courses once they've had practice on the syntax and structure of functions and function-calls. While this is the case, they may still fail to write effective code for tasks if they have not seen similar tasks or task-compositions in their coursework, and thus do not have patterns to retrieve to actualize the tasks. With decomposing-tasks, while students may identify the tasks in the problem, they may still fail to actualize those tasks if they have not seen concrete patterns for doing so. The knowledge of applicable patterns seems to be the binding factor between being able to identify the tasks and actualizing those tasks in code – if a student can identify tasks, but do not have patterns to retrieve for those tasks, they may not be able to implement those tasks or implement the composition of those tasks.
H. Course-specific design-practices used across the two problems 1. Did they use similar/different practices?	1. See F.3. on the use of data-definitions.