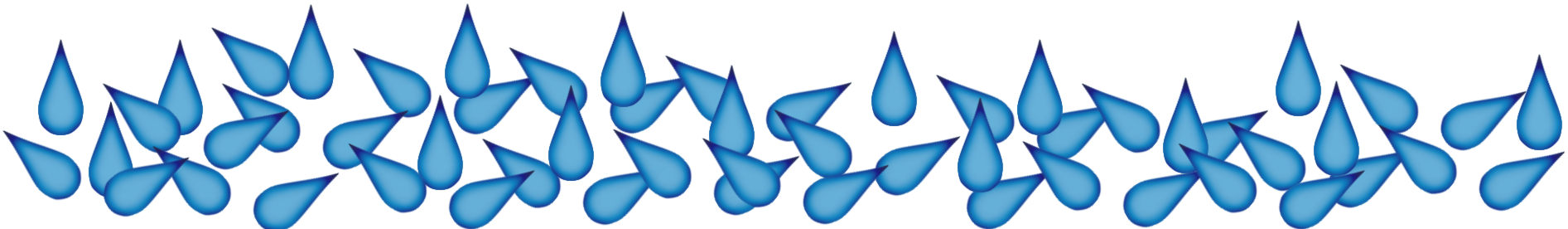
Sometimes, Rainfall Accumulates:

Talk-alouds with novice functional programmers

Kathi Fisler (Brown University)

Francis Castro (WPI)



Rainfall (Soloway): *compute average of non-negative inputs that occur prior to -999*

Long history of challenging students:
many subtasks (count, sum, rem negatives, …),
which need to be integrated into a program

What's Known: Given a programming problem, people recall related schema (if possible) and adapt it to problem

General pattern used to solve a problem
(i.e., loop over data and maintain a counter)

What if people have learned multiple schemas? How do they navigate the choice between them?

As part of another (larger) study on student planning practices, we asked students to talk aloud while they solved Rainfall

Students had 30 minutes, after which we interviewed them about how they approached the problem (relative to what they had learned in the course)

Our paper presents narratives of four of these sessions, and our observations

Course Context

A CS1 course, based on *How to Design Programs*, with programming in Racket (Scheme cousin)

Write examples before writing code

```
;; rainfall : list-of-number -> number
;; compute average of non-neg nums before -999
(define (rainfall alon)
  (cond [(empty? alon) ...]
        [(cons? alon)
         ... (first alon)
         ... (rainfall (rest alon)) ...]))
```

template: code structure should follow input type

Schema for programs that take lists

Naïve template breaks down on Rainfall

```
;; rainfall : list-of-number -> number
;; compute average of non-neg nums before -999
(define (rainfall alon)
  (cond [(empty? alon) ...]
        [(cons? alon)
         ... (first alon)
         ... (rainfall (rest alon)) ...]))
```

Rainfall is a more advanced problem

Template computes average from average of "rest" of list and one additional element

Course hadn't discussed this

Template applies for functions that involve only one "traversal" task

Valid Approaches to Rainfall

```
;; rainfall : list-of-number number number -> number
;; compute average of non-neg nums before -999
(define (rainfall alon sum count)
  (cond [(empty? alon) (/ sum count)]
        [(cons? alon)
         (if (>= (first alon) 0)
```

Use Accumulators

```
;; rainfall : list-of-number -> number
;; compute average of non-neg nums before -999
(define (rainfall alon)
  (let* ([cleannums (filter non-neg? alon)]
         [total (sum cleannums)]
         [count (length cleannums)])
    (cond [(empty? cleannums) -1]
          [(cons? cleanums) (/ total count)])))

(define (sum alon)
  (cond [(empty? alon) 0]
        [(cons? alon) (+ (first alon)
                         (sum (rest alon)))]))
```

Course had covered accumulators and all of the "known" functions

Decompose and Use Known Functions

Students had seen multiple ways to approach Rainfall
(though not traversal-based decomposition)

Which would they try first?

Would they change approaches mid-stream?

Haven't found prior research on how students navigate multiple schemas

Interview (post talk-aloud) Questions

- Was the problem statement clear when you read it?
- What did you think of doing first? Were you reminded of a construct [or] general structure of solution that you thought would be useful?
- Did you feel stuck at any point while working?
- Describe your code's approach to solving the problem.
- Were the program design techniques taught in class helpful to you when solving this problem?
- Did you use program design techniques not taught in class?

Reminder: The Rainfall problem

Compute the average of non-negative numbers that occur prior to -999 in an input list. If you cannot compute the average for whatever reason, return -1.

(our wording is in the paper; it provided additional context for the problem)

Student #1

writes entire template

returns -1 on empty list

add accumulator to track sum

changes empty answer to accumulator value

trashes on where to put the division that she knows is part of average

```
(define (rainfall alon acc)
  (cond
    [(empty? alon) acc]
    [(cons? alon)
     (if (< (first alon) 0)
         (rainfall (rest alon) acc)
         (rainfall (rest alon)
                   (+ 1 acc))
         (rainfall (rest alon) acc))]))
```

follows class example

follows schema

off any schema

"I thought accumulator would be useful because every time it finds another positive value [...] the average changes because the bottom number would keep getting bigger. So the accumulator would keep adjusting to that."

Student #2

writes beginning of template

returns -1 on empty list

add accumulator, no clear purpose

writes function to filter out the negative nums

doesn't integrate negatives, realizes can make second accumulator

working syntactically

follows schema

generalizes schema to new situation

```
(define (rainfall alon acc times)
  (cond
    [(empty? alon) acc]
    [(cons? alon)
     (if (> (first alon) -999)
         (rainfall (rest alon)
                   (/ (+ (first alon) acc)
                      times)
                   (+ 1 times))
         (rainfall (rest alon) acc times))]))
```

The average is going to take into account not only how much was added […] but also how many times it was added. […] it almost seems like I would use an accumulator to show how many times I've actually gotten through that. […] And then also another accumulator – we'll change it to something different – to times

Students who copy-and-paste the template (as HtDP recommends for beginners) get more stuck than those who recall the template and write it down "as they go".

```
;; rainfall : list-of-number -> number
;; compute average of non-neg nums before -999
(define (rainfall alon)
  (cond [(empty? alon) ...]
        [(cons? alon)
         ... (first alon)
         ... (rainfall (rest alon)) ...]))
```

Students who articulated only the syntactic schema of accumulators, but not the underlying concept, struggled to adapt them to the needs of Rainfall.

Rainfall needs two accumulators
(one for sum, one for count),
but the course had only shown
examples with one accumulator

Students who connected accumulator parameters or parts of their code to specific tasks, and maintained those connections through the schema switch, produced more correct code.

Reinforces the value of thinking in terms of plans and subtasks; takes students from syntactic to semantic work

Students had not understood that each sub-task that traverses a list needs its own function or accumulator parameter.

Takeaways

- Reading transcripts highlighted *syntactic* nature of schemas, even for good students
- Stronger students had some sense of when to use specific schemas, but less than we hoped
  - generalizing schemas (i.e., multiple accumulators) appears more subtle than we expected
- Rainfall continues to be a deceptively interesting problem

Instructors, carry your umbrella!
(we are in Seattle, after all ...)