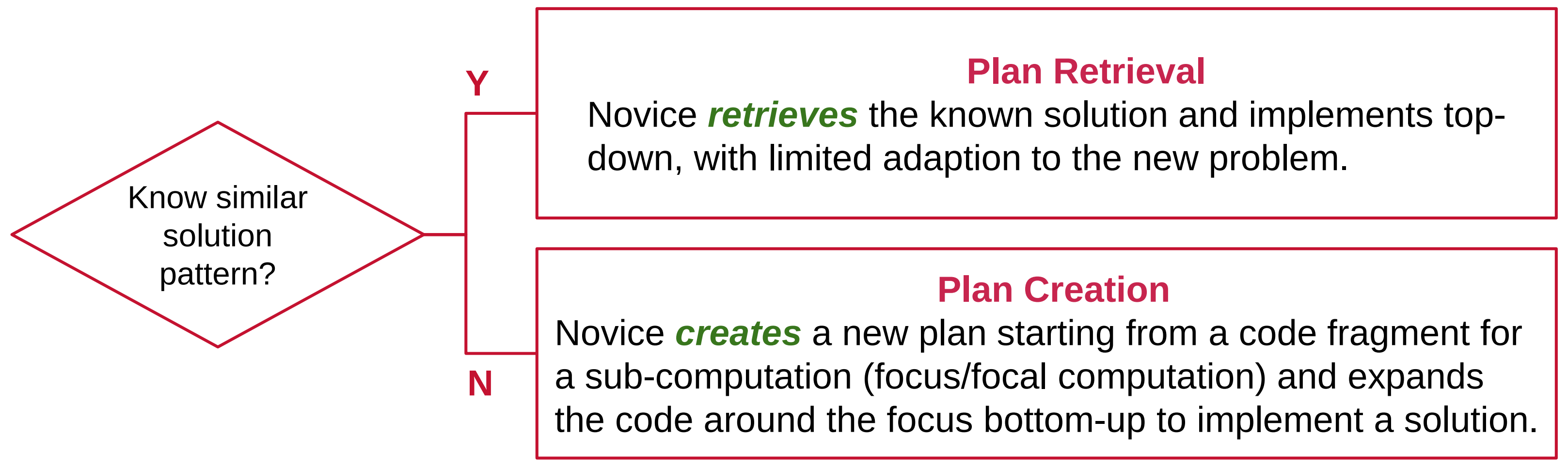
On the Interplay Between Bottom-Up and Datatype-Driven Program Design

Francisco Castro, fgcastro@wpi.edu
Advisor: Kathi Fisler, kfisler@wpi.edu
Department of Computer Science

Abstract

When students are faced with a programming problem unlike any they have solved before, prior research suggests that they develop code backwards from essential computations in the problem. Some curricula, however, teach students to first write scaffolding code based on the type of the input data. How do these two approaches interact? We gave CS1 students who were taught to write scaffolding code a programming problem unlike any they had seen before. We found that while students put essential computations into the scaffolds, they often overuse affordances of the scaffolds in ways that lead to plan-composition errors. We propose that steering students away from on-the-fly decomposition while programming could help avoid some of these errors.

Models of Novice Programmer Behavior



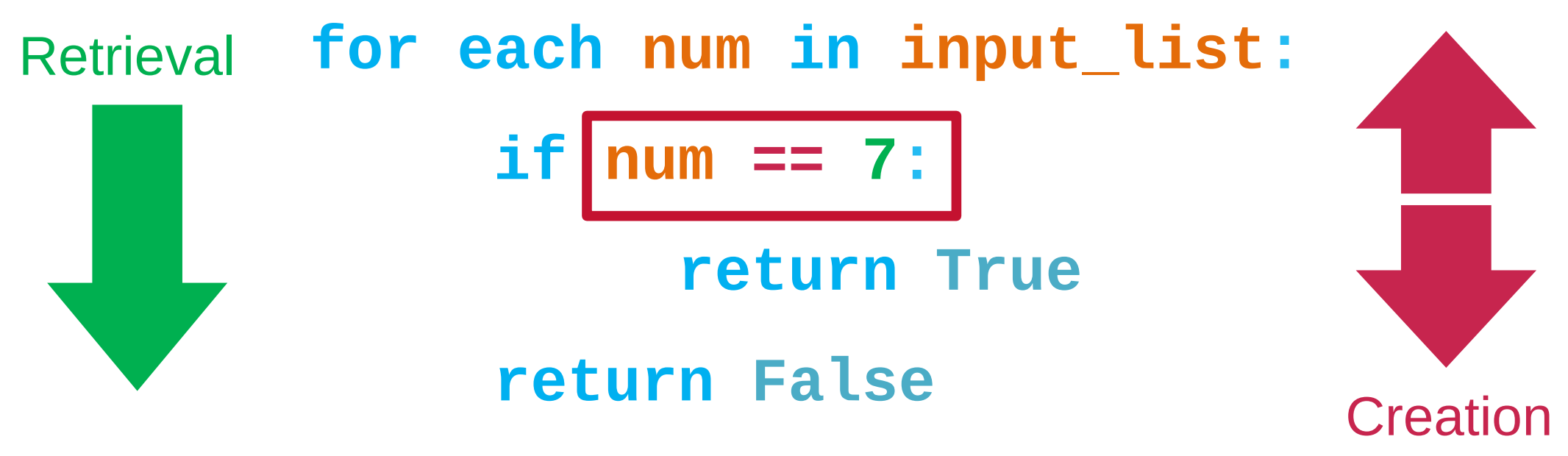
Plan Retrieval
Novice ***retrieves*** the known solution and implements top-down, with limited adaption to the new problem.

Plan Creation
Novice ***creates*** a new plan starting from a code fragment for a sub-computation (focus/focal computation) and expands the code around the focus bottom-up to implement a solution.

- Novices rely heavily on previously learned program plans, examples, or solutions, fitting learned solutions into the context of new problems [2,4]
- The focal expansion model identifies the states of (1) plan retrieval and (2) plan creation to describe novice programmer behavior when encountering a programming problem [3]

Program Design

Focal Expansion



```
for each num in input_list:
    if num == 7:
        return True
    return False
```

Problem:
Determine if a list of numbers contains 7

Datatype-Driven

```
(define (containsNum? alist)
  (cond [(empty? alist) ... ]
        [(cons? alist) ... (first alist)
                       ... (containsNum? (rest alist))]))

(define (containsNum? alist)
  (cond [(empty? alist) false]
        [(cons? alist) (or (= 7 (first alist))
                           (containsNum? (rest alist)))]))
```

How to Design Programs (HTDP) Core Idea:
Design programs from the structure of the input data

Methodology

Research Questions

1. When do HTDP-trained students use templates?
2. How do focal computations manifest in HTDP programs?
3. How and when do HTDP students integrate focal computations into existing code?

Data Collection

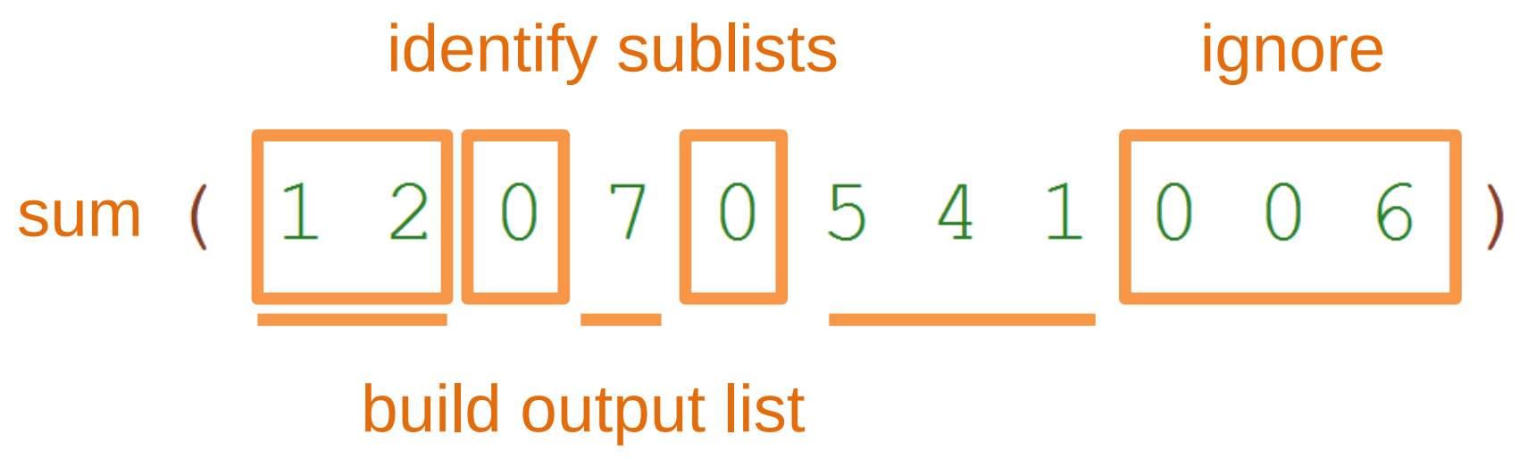
- Spring 2015 CS1 course using HTDP in Racket
- Participants worked on the Adding Machine problem during a weekly lab session
- Video captured activity within the IDE window
- 25 (of 138) submissions analyzed

Problem: Adding Machine

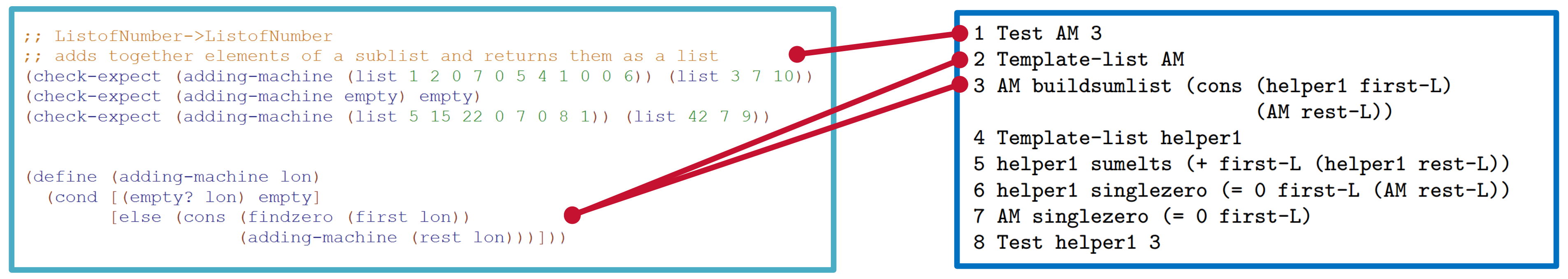
Design a program called **`adding-machine`** that consumes a list of numbers and produces a list of the sums of each non-empty sublist separated by zeros. Ignore input elements that occur after the occurrence of two consecutive zeros.

```
Input:  (list 1 2 0 7 0 5 4 1 0 0 6)

Output: (list 3 7 10)
```



Data Coding

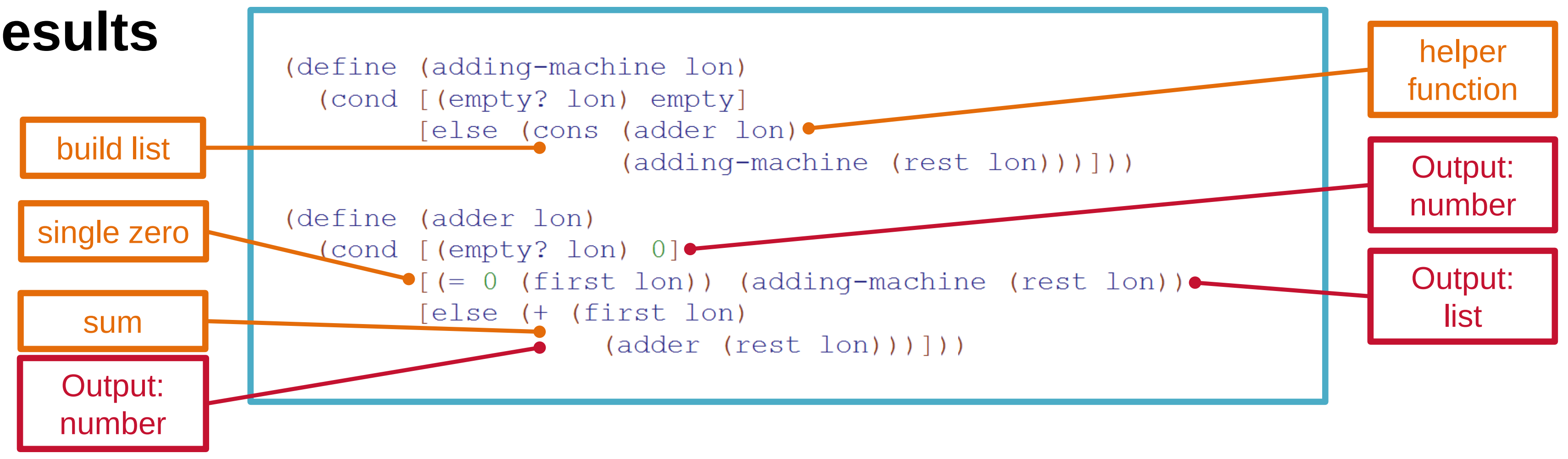


```
;; ListofNumber->ListofNumber
;; adds together elements of a sublist and returns them as a list
(check-expect (adding-machine (list 1 2 0 7 0 5 4 1 0 0 6)) (list 3 7 10))
(check-expect (adding-machine empty) empty)
(check-expect (adding-machine (list 5 15 22 0 7 0 8 1)) (list 42 7 9))


(define (adding-machine lon)
  (cond [(empty? lon) empty]
        [else (cons (findzero (first lon))
                    (adding-machine (rest lon)))]))
```

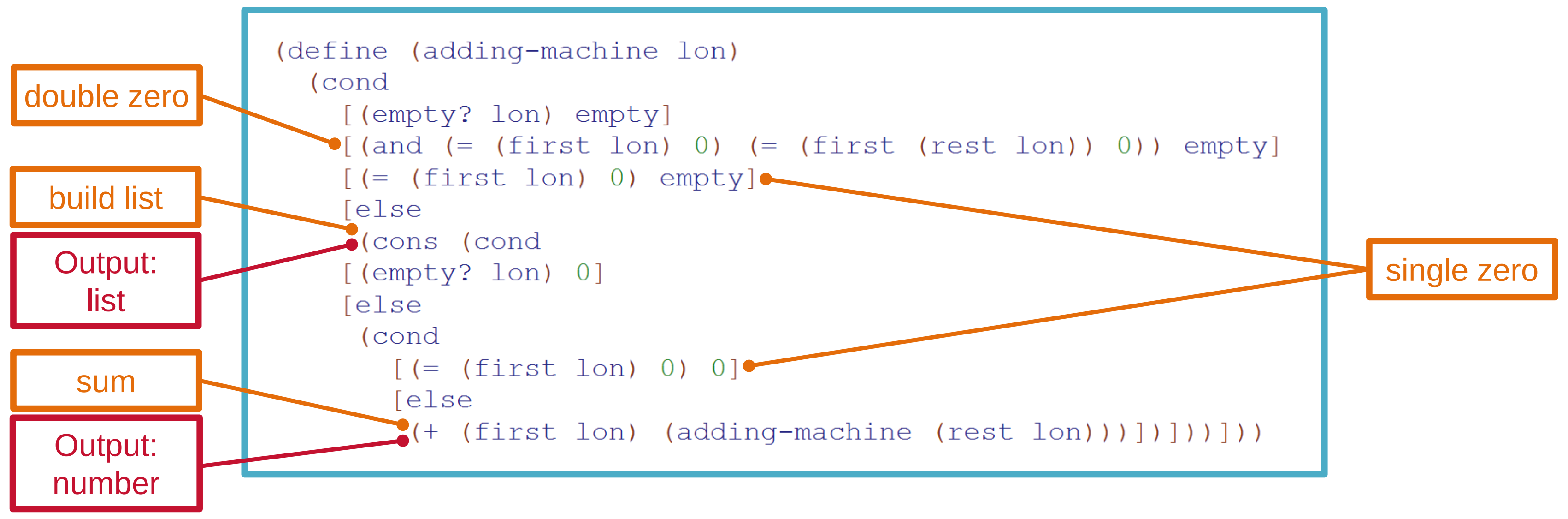
```
1 Test AM 3
2 Template-list AM
3 AM buildsumlist (cons (helper1 first-L)
                        (AM rest-L))
4 Template-list helper1
5 helper1 sumelts (+ first-L (helper1 rest-L))
6 helper1 singlezero (= 0 first-L (AM rest-L))
7 AM singlezero (= 0 first-L)
8 Test helper1 3
```

Results

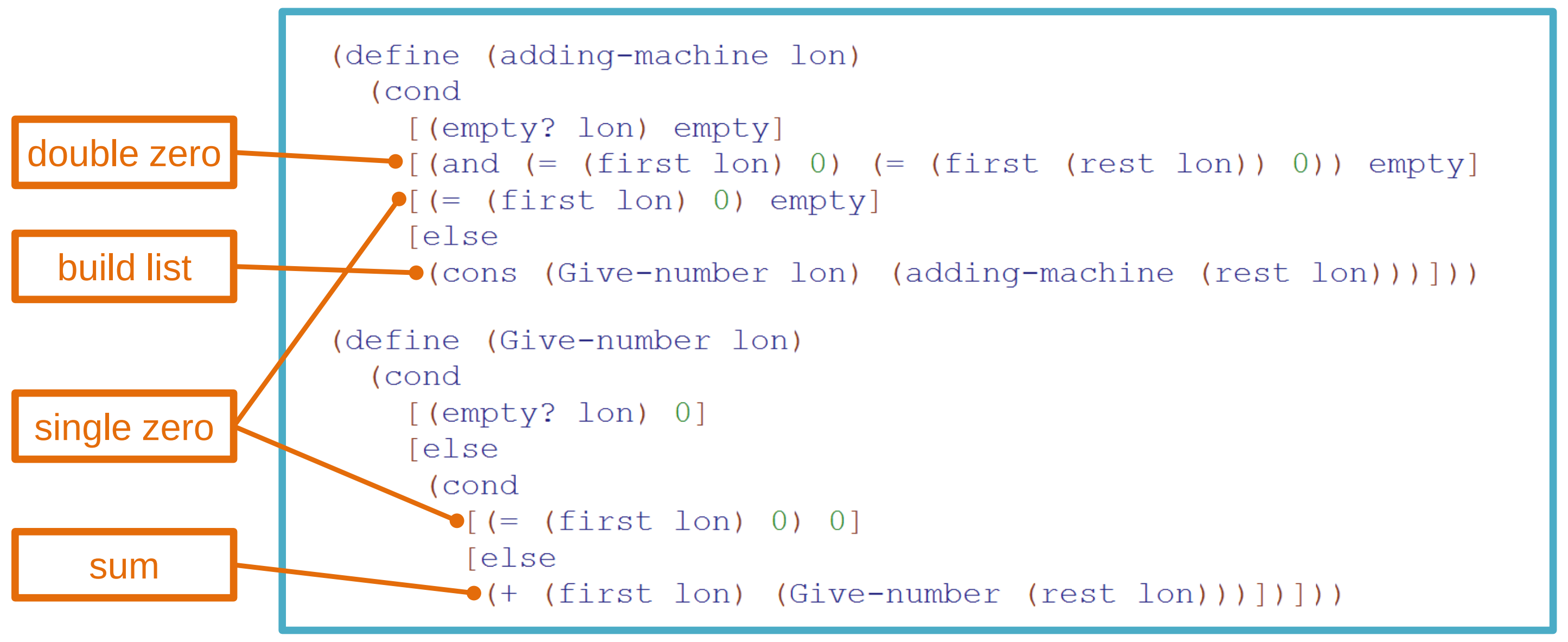


```
(define (adding-machine lon)
  (cond [(empty? lon) empty]
        [else (cons (adder lon)
                    (adding-machine (rest lon)))]))

(define (adder lon)
  (cond [(empty? lon) 0]
        [(= 0 (first lon)) (adding-machine (rest lon))]
        [else (+ (first lon)
                 (adder (rest lon)))]))
```



```
(define (adding-machine lon)
  (cond
    [(empty? lon) empty]
    [(and (= (first lon) 0) (= (first (rest lon)) 0)) empty]
    [(= (first lon) 0) empty]
    [else
     (cons (cond
     [(empty? lon) 0]
     [else
      (cond
        [(= (first lon) 0) 0]
        [else
        (+ (first lon) (adding-machine (rest lon)))])]))]))
```



```
(define (adding-machine lon)
  (cond
    [(empty? lon) empty]
    [(and (= (first lon) 0) (= (first (rest lon)) 0)) empty]
    [(= (first lon) 0) empty]
    [else
     (cons (Give-number lon) (adding-machine (rest lon)))]))

(define (Give-number lon)
  (cond
    [(empty? lon) 0]
    [else
     (cond
      [(= (first lon) 0) 0]
      [else
       (+ (first lon) (Give-number (rest lon)))])]))
```

Findings

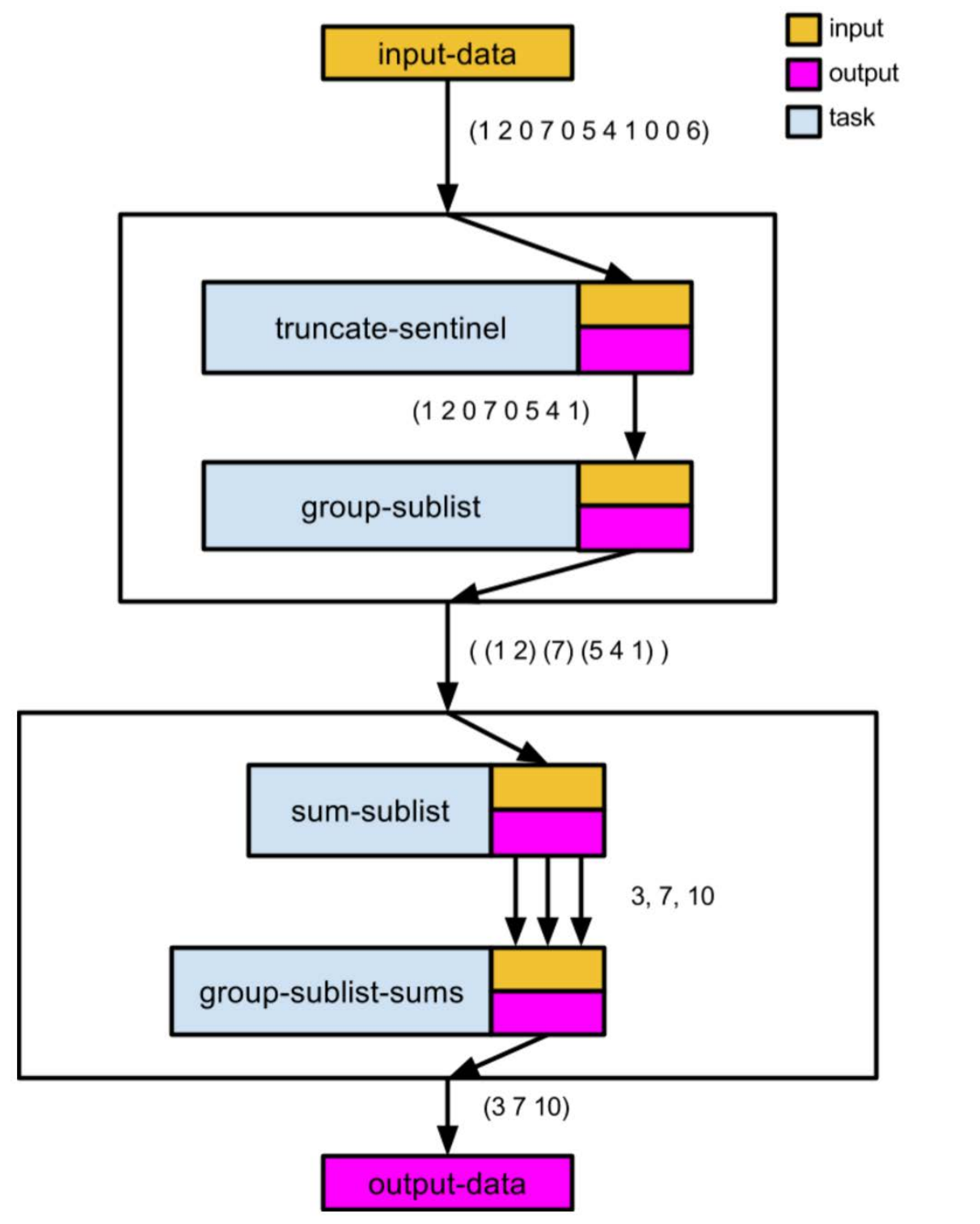
- Evidence of HTDP template use, development of focals, and task decomposition
- Students created helpers but failed to use them to effectively decompose the problem, attempting various task combinations and replicating tasks within and across functions
- Students struggled with problem decomposition and plan composition, resulting in output inconsistencies and errors

Key Takeaways

- Data suggests that students largely work through problem tasks
- Students retrieved plans in the form of (a) operational expressions and (b) entire functions
- Key issues: on-the-fly problem decomposition around existing code and the retrieval of contexts that aren't well suited to the problem
- Students struggled to decompose the problem and compose plans – they were not taught a systematic process for doing these

Future Work

- Develop pedagogical interventions that teach principles of problem decomposition and plan composition
- Use concrete examples to work out problem decompositions
- Teach data-centric principles for programming – i.e. data transformation to make subsequent computations easier; plan dependencies to work out plan composition



References

[1] M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi. How to Design Programs. MIT Press, 2001.
[2] P. L. Pirolli and J. R. Anderson. The role of learning from examples in the acquisition of recursive programming skills. Canadian Journal of Psychology, 39(2):240–272, 1985.
[3] R. S. Rist. Knowledge creation and retrieval in program design: A comparison of novice and intermediate student programmers. Hum.-Comput. Interact., 6(1):1–46, Mar 1991.
[4] J. C. Spohrer and E. Soloway. Novice mistakes: Are the folk wisdoms correct? Commun. ACM, 29(7):624–632, July 1986.

Acknowledgments

We thank Joe Beck for letting us collect data in his course and Mike Clancy for pointing us to the Adding Machine problem. This research is partially funded by the US NSF under grant number CCF1116539.