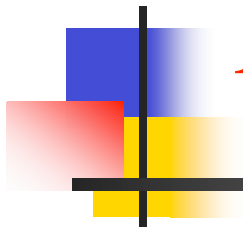


# Solving More Complex List Problems





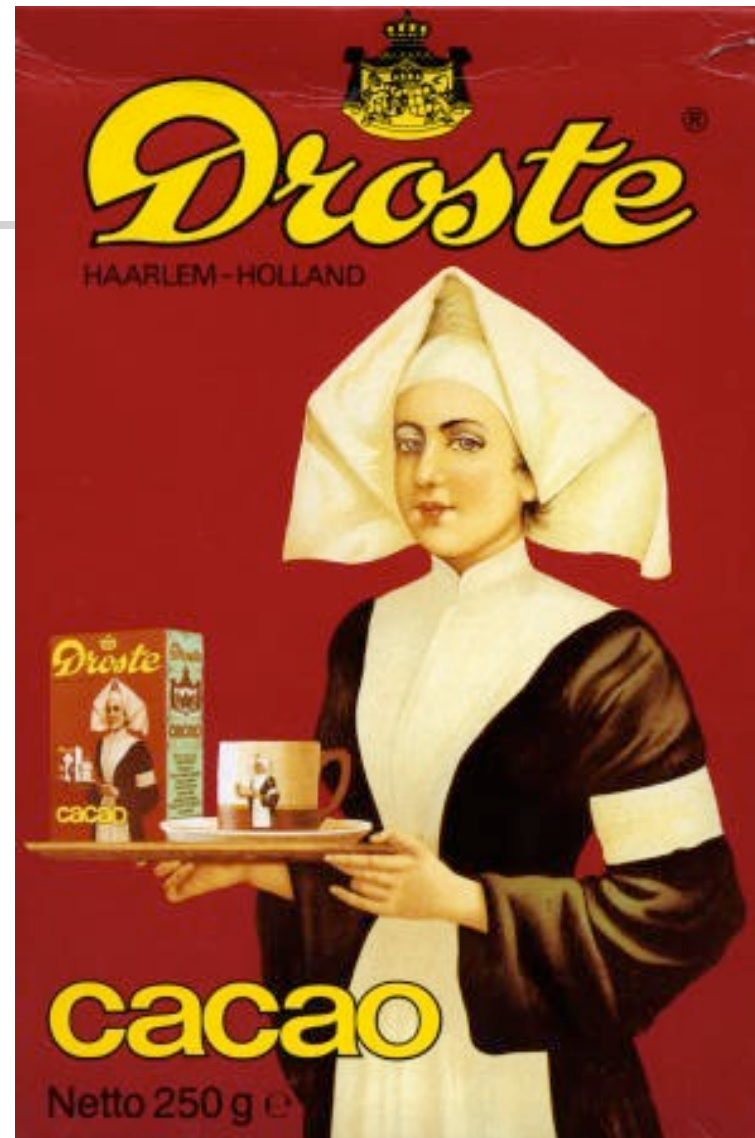
## Feedback from today's quiz

---

- “I'm somewhat unclear as to how lists-within-lists work for combining list and cons...”
- “One aspect of recursion is still a bit difficult to grasp: How does the program know what to do with the function name for the last part of the else branch?”

## Second Question

- How can we draw this picture in a finite amount of time?





## Second Question

---

- Consider this simple recursive definition

```
(define (f n)
  (cond [(= 0 n) 1]
        [else (* n (f (- n 1)))]))
```

- We can understand this definition “bottom-up”

$(f\ 0) \dots = 1$  // from first branch of `cond`

$(f\ 1) \dots =$  // from second branch and  $(f\ 0)$   
 $(* 1 (f\ 0)) = (* 1 1) = 1$

$(f\ 2) \dots =$  // from second branch and  $(f\ 1)$   
 $(* 2 (f\ 1)) = (* 2 1) = 2$

$(f\ 3) \dots = 6, (f\ 4) \dots = 24, (f\ 5) \dots = 120, \dots$



## Second Question

---

- This builds an infinite “table” of inputs and outputs
- In Scheme, we don’t want to build infinite tables (even though it is a nice mathematical exercise)
- Instead, the **reduction rules** compute the parts of the table we need (and only the parts we need)

- Here is a (summarized) reduction for  $(f\ 5)$

$$\begin{aligned}(f\ 5) &= \dots = (*\ 5\ (f\ (-\ 5\ 1))) = (*\ 5\ (f\ 4)) \\ &\dots = (*\ 5\ (*\ 4\ (f\ (-\ 4\ 1)))) \\ &\dots = (*\ 5\ (*\ 4\ (*\ 3\ (*\ 2\ (*\ 1\ (f\ 0)))))) \\ &\dots = (*\ 5\ (*\ 4\ (*\ 3\ (*\ 2\ (*\ 1\ 1)))) \\ &\dots = 120\end{aligned}$$

- Good test question: fill in the ...’s



## Second Question

---

- The real challenge is in how to
  - Go from **problem** to recursive solution
- Initially, it will look like magic. That's OK.
  - Problem: “The factorial of a number is the product of all the numbers from 1 upto that number”
  - Magic insight: “If I know factorial for  $n$ , I know how to compute factorial for  $n+1$ ”
- The good news is, eventually
  - You will learn “to divide and concur”
  - Only a small number of “recursion” patterns are used



# Today's Goals

---

- A new tool: List abbreviations
- Bigger list processing problems
- Example: Sort
- Example: Arrangements (permutations)



# List Abbreviations

---

- We use lists a lot when we program
- Using `empty` and `cons` is verbose
- Example:
  - `(list 1 2 3)`
  - `(cons 1 (cons 2 (cons 3 empty)))`
- Note that
  - `empty` disappears
  - `List` is not a function that we can write
  - Quantitatively, how much easier is it to write lists now?



# List Abbreviations

---

- Example
  - `'((1 2) (3 four))`
  - `(list (list 1 2) (list 3 'four))`
  - `(cons (cons 1 (cons 2 empty)) (cons (cons 3 (cons 'four empty)) empty))`
- Note that
  - Changes the way parentheses are treated
  - Nesting is quite subtle
  - What happens if we write `'(list 1 2 3)` ?



# Solving more complex problems

---

- Goal:
  - To divide problem into smaller part
  - Not to solve the smaller parts (yet)
- Functional decomposition
  - A technique for program analysis
  - Look for what should be a function
  - Sign: A dependency



## Example: Insert Sort

---

- Sort: [number] -> [number]
- Example:
  - Given ' ( 3 5 4 2 1 )
  - Return ' ( 1 2 3 4 5 )
- Example:
  - Given ' ( 100 3 400 )
  - Return ' ( 3 100 400 )
- Can we devise a method for doing this?



## Sort (in class exercise)

---

- The template gives us **specific questions** that we need to answer to solve the problem
- It will not always work. But often it does.
  - (There are problems that require other templates)
- In the case of the sorting problem, answering the template questions leads us to recognize that we would solve the problem if only we had a function `insert...`
- Answering the questions also gave us the code for the main function `sort`



## Insertion (in class exercise)

---

- Using the template is a **huge** help in solving this problem. Here is the code we got by following it:

```
(define (insert element lst)
  (cond
    [(empty? lst) (cons element empty)]
    [else (cond
              [(< element (first lst))
               (cons (element lst))]
              [else (cons (first lst)
                          (insert element
                                (rest lst)))]))])
```



## Example: Arrangements

---

- Arrange: [symbol] -> [[symbol]]
- Example:
  - Given ' ( a b )
  - Return ' ( ( a b ) ( a b ) )
- Example:
  - Given ' ( a b c )
  - Return ' ( ( a b c ) ( c a b ) ( b c a )  
( c b a ) ( a c b ) ( b a c ) )



## For Next Class

---

- Homework due Monday
- Reading:
  - Chapters 14 + companion notes
  - Trees!
- Quiz:
  - Chapter 14