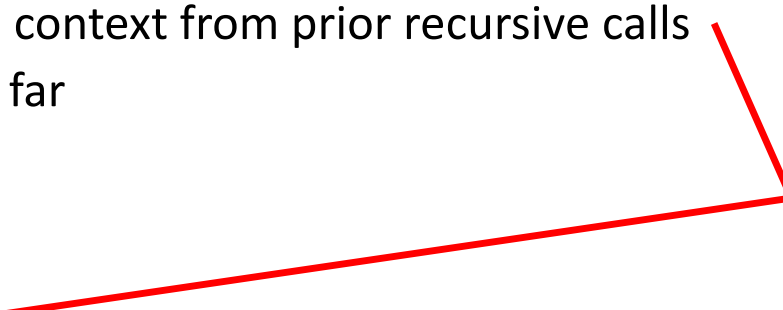


Accumulators

- accumulate information from prior recursive calls
 - three categories
 - preserve context from prior recursive calls
 - result so far
 - worklist
- 
- lecture 18 (previous)
- this lecture
- The diagram consists of two red lines. One line starts at the text 'lecture 18 (previous)' and points to the item 'preserve context from prior recursive calls'. The other line starts at the text 'this lecture' and points to the item 'worklist'.

Roadmap

- The next four lectures
 - forms of data: trees and **graphs**
 - recursion: structural non-tail and tail and generative
 - accumulators
 - path in data: previous, upper, lower, pnum, **path**, **visited**...
 - rsf (result so far)
 - path in tail recursion: visited, count, ...
 - worklist
 - **tandem worklist**

l18 l19 l20 l21

```

(define (sequence? lon0)
  ;; prev is Natural
  ;; invariant: the element of lon0 immediately before (first lon)
  ;; (sequence? (list 2 3 4 7 5))

  ;; (sequence? (list 3 4 7 5) 2)
  ;; (sequence? (list 4 7 5) 3)
  ;; (sequence? (list 7 5) 4) ==> false
  (local [(define (sequence? lon prev)
            (cond [(empty? lon) true]
                  [else
                   (if (= (first lon) (+ 1 prev)) ;exploit (use)
                       (sequence? (rest lon)
                                   (first lon)) ;preserve
                       false))]])

    (if (empty? lon0) ;if original list is empty, we can't
        true ;initialize accumulator, so special case
        (sequence? (rest lon0)
                    (first lon0)))) ;initialize

```

an expression is in tail-position when no surrounding expression in that function is waiting to operate on the value of the expression

tail position → (define (mumble x)
 (* (+ 1 x) 2))

not in tail
position →

```
;; for if, cond and local expressions,  
;; IF THE WHOLE EXPRESSION IS IN TAIL POSITION,  
;; then  
;; questions are never in tail position  
;; answers are in tail position  
;; body is in tail position
```

```
(if <question>  
    <true answer>  
    <false answer>)
```

```
(cond [<question> <answer>  
      ...])
```

```
(local [<definition> ...]  
      <body>)
```

definitions

- an expression is in tail-position when no surrounding expression in that function is waiting to operate on the value of the expression
- a function is tail-recursive when ALL recursive calls are in tail position
- mutually-recursive functions are tail recursive when ALL recursive and mutually recursive calls are in tail position
- in tail recursive functions, base cases produce the final result

```
(define (fn-for-lox lox0)
  (cond [(empty? lox) empty] ;base is empty
        [else                ;combination is cons
         (cons (first lox)
               (fn-for-lox (rest lox)))]))

not in tail position
```



```
(define (rev lox0)
  (local [(define (fn-for-lox lox rsf)
            (cond [(empty? lox) empty]
                  [else
                   (fn-for-lox (rest lox)
                               (cons (first lox)
                                     rsf))])]
    (fn-for-lox lox0 empty)))

tail position
```

Diagram illustrating the tail position of recursive calls in two Racket functions:

- Top Function:** `(define (fn-for-lox lox0) ...)`. The recursive call `(fn-for-lox (rest lox))` is **not in tail position**. The combination `(cons (first lox) ...)` is highlighted with a dashed red circle, indicating it is the **combination** (cons).
- Bottom Function:** `(define (rev lox0) ...)`. The recursive call `(fn-for-lox (rest lox) ...)` is **in tail position**. The combination `(cons (first lox) rsf)` is highlighted with a dashed red circle, indicating it moves into the **rsf update**.

Converting recursive call wrapped in combination to tail call

```
(... (first lox) (fn-for-lox (rest lox)))
```

```
(fn-for-lox (rest lox) (... (first lox)))
```

```
(fn-for-lox (rest lox) (... (first lox) rsf))
```

```
(define (rev lox0)
```

```
  (local [(define (fn-for-lox lox)
             (cond [(empty? lox) (...)]
                   [else
                    (... (first lox)
                        (fn-for-lox (rest lox))))]))]
```

```
  (fn-for-lox lox0)))
```

```
(define (rev lox0)
```

```
  ;; rsf is (listof X)
```

```
  ;; all elements of lox0 before (first lox), in reverse order
```

```
  (local [(define (fn-for-lox lox rsf)
```

```
            (cond [(empty? lox) rsf]
```

```
                  [else
```

```
                  (fn-for-lox (rest lox) (cons (first lox) rsf))]))]
```

```
  (fn-for-lox lox0 empty)))
```

produce rsf at end

combination

initialize rsf (base case result)


```
:: QUESTION 1 [45 seconds]
::
;; Is the call to positive? in tail position?
```

```
(define (positive-only lon)
  (cond [(empty? lon) empty]
        [else
         (if (positive? (first lon))
             (cons (first lon)
                   (positive-only (rest lon)))
             (positive-only (rest lon)))]))
```

```
:: A. Yes
:: B. No
```

;; QUESTION 2 [30 seconds]

;;

;; Is the recursive call to positive-only labeled (1) in tail position?

```
(define (positive-only lon)
  (cond [(empty? lon) empty]
        [else
         (if (positive? (first lon))
             (cons (first lon)
                   (positive-only (rest lon))) ;(1)
             (positive-only (rest lon)))])) ;(2)
```

;; A. Yes

;; B. No

```
;; QUESTION 3 [20 seconds]
```

```
;;
```

```
;; Is the recursive call to positive-only labeled (2) in tail position?
```

```
(define (positive-only lon)
  (cond [(empty? lon) empty]
        [else
         (if (positive? (first lon))
             (cons (first lon)
                   (positive-only (rest lon))) ;(1)
             (positive-only (rest lon))))]) ;(2)
```

```
;; A. Yes
```

```
;; B. No
```

```
;; QUESTION 4 [40 seconds]
;;
;; Is positive-only tail-recursive?
```

```
(define (positive-only lon)
  (cond [(empty? lon) empty]
        [else
         (if (positive? (first lon))
             (cons (first lon)
                   (positive-only (rest lon)))
             (positive-only (rest lon))))]))
```

```
;; A. Yes
;; B. No
```

;; QUESTION 5

;;

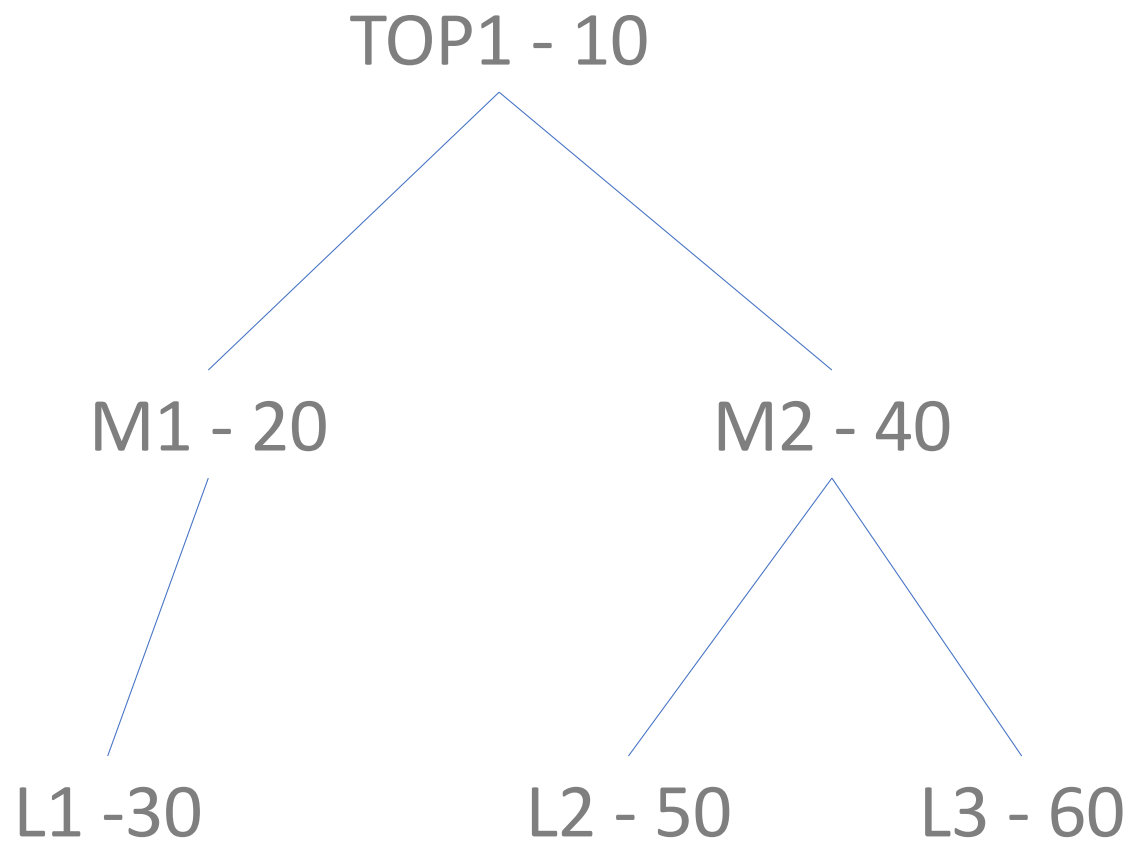
;; Is positive-only tail-recursive? [40 seconds]

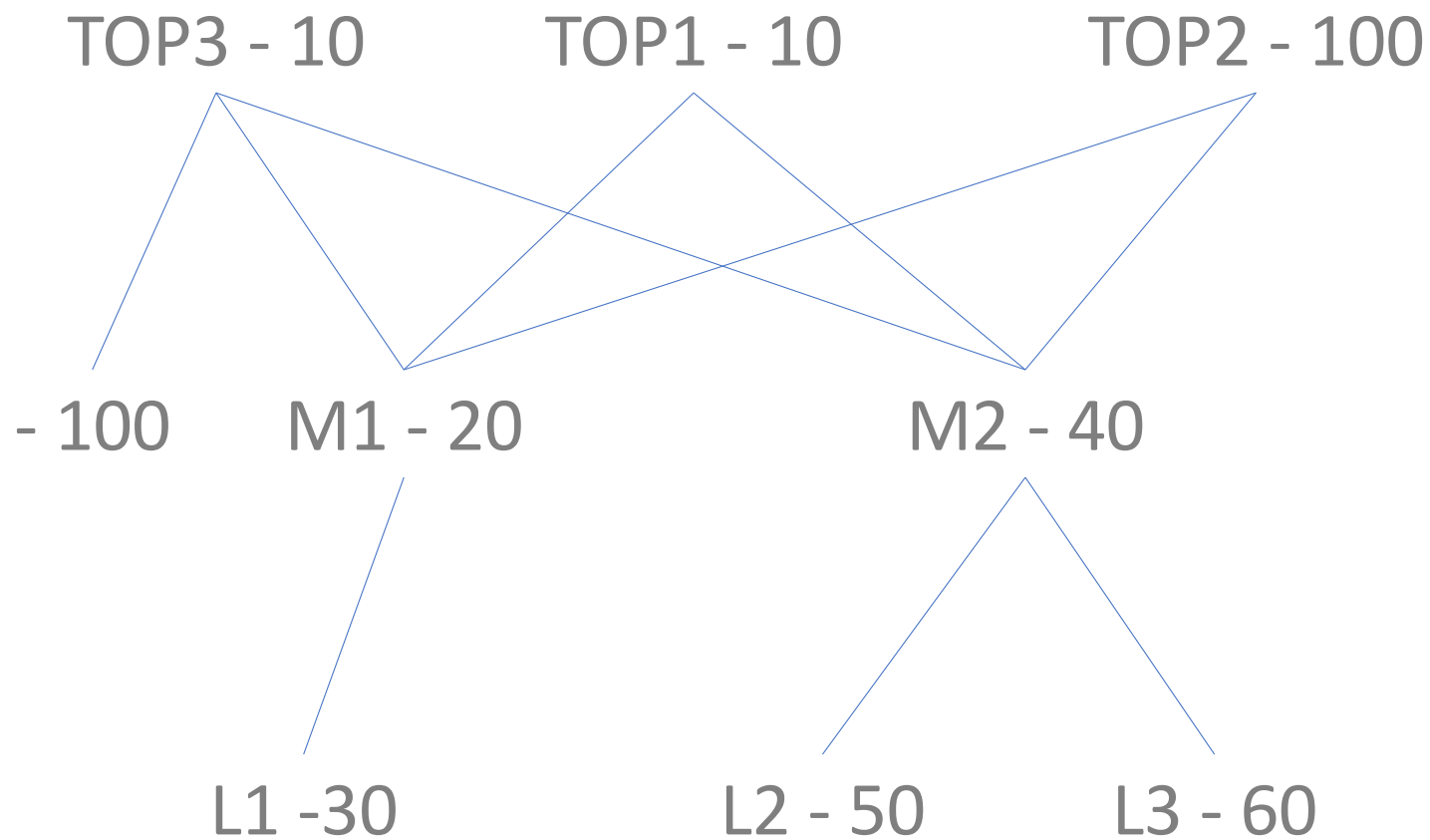
```
(define (positive-only lon0)
  (local [(define (positive-only lon rsf)
            (cond [(empty? lon) (reverse rsf)]
                  [else
                   (if (positive? (first lon))
                       (positive-only (rest lon) (cons (first lon) rsf))
                       (positive-only (rest lon) rsf))]))]
    (positive-only lon0 empty)))
```

;; A. Yes

;; B. No

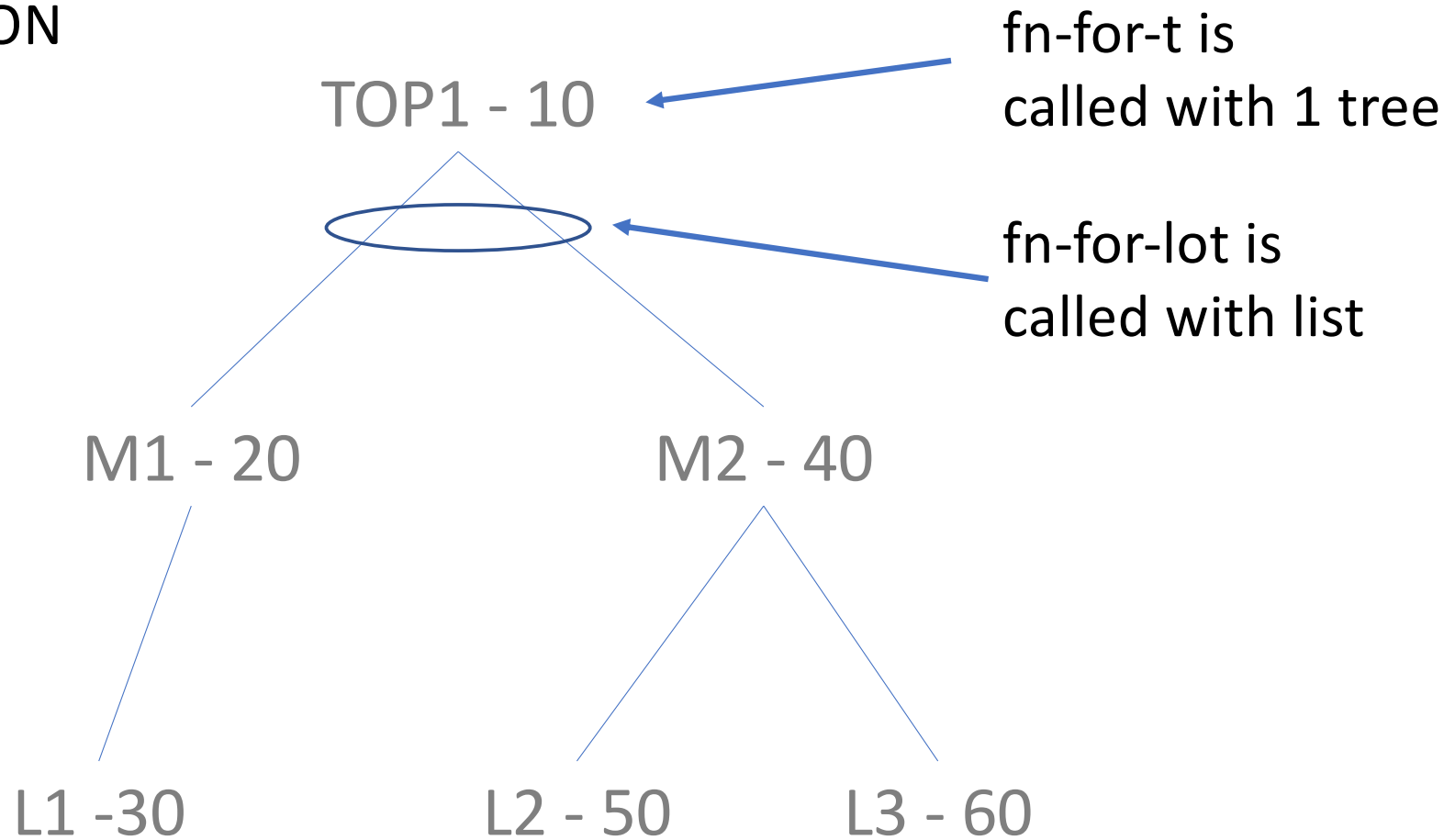
Example Tree





2 more
Example Trees

NOTATION




```

(define (top->bot-sorted? t0)
  ;; pnum is Integer; immediate parent node's number
  ;; **parent means the parent in the tree **
  (local [(define (fn-for-t t pnum)
            (local [(define number (node-number t)) ;unpack the fields
                    (define subs (node-subs t))] ;for convenience
              (if (> number pnum)
                  (fn-for-lot subs number)
                  false)))

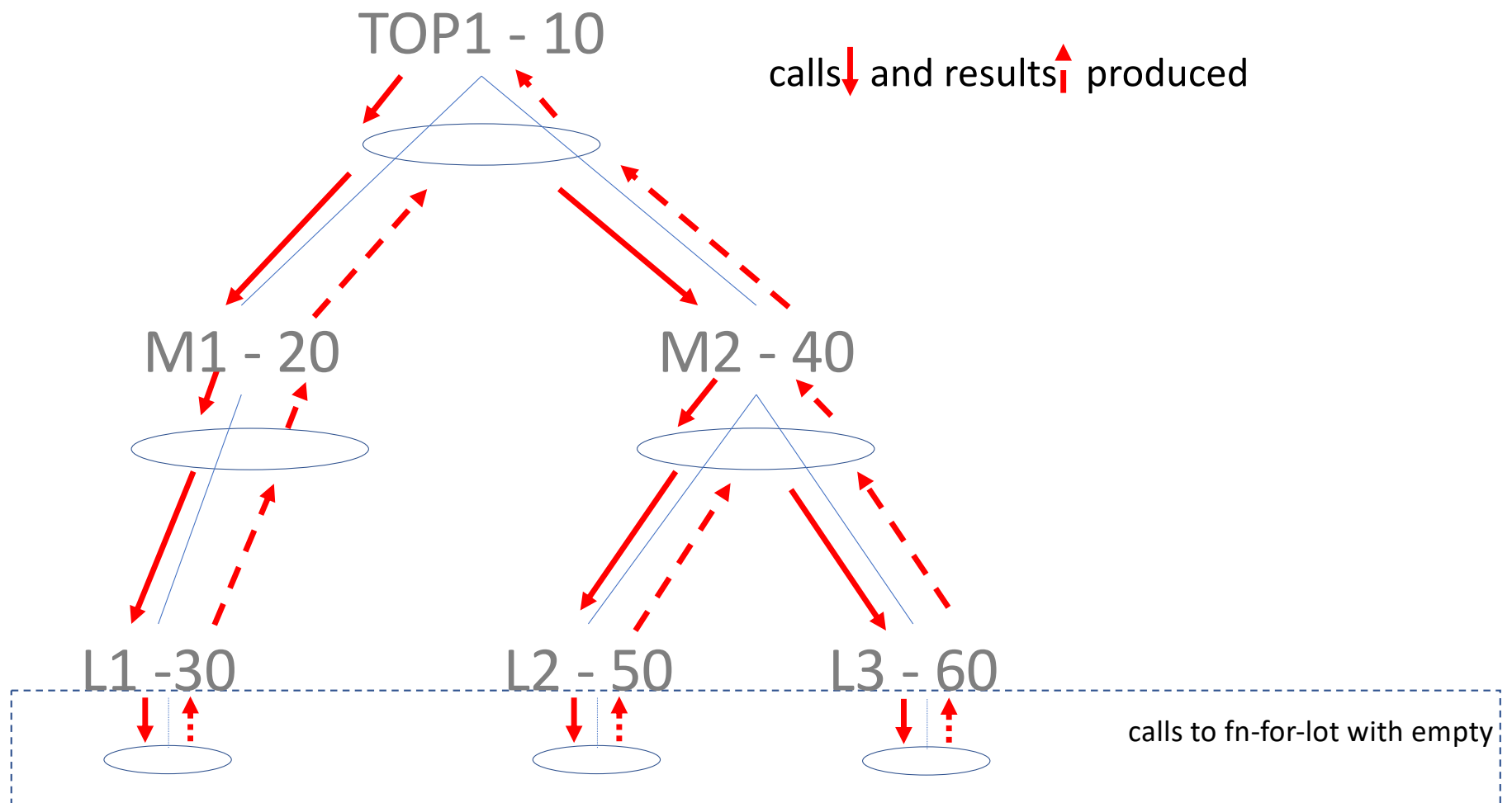
            (define (fn-for-lot lot pnum)
              (cond [(empty? lot) true]
                    [else
                     (and (fn-for-t (first lot) pnum)
                          (fn-for-lot (rest lot) pnum))]))])

  (fn-for-lot (node-subs t0) (node-number t0)))

```

typical structural recursion

calls↓ and results↑ produced



What's in the past depends on the recursion

- tail recursion means current call can have all the preceding context
 - it can produce answer directly
- in a tree
 - ordinary recursion can carry context of what is above current call
 - but tail recursion is required to carry context of what is above and to the LEFT

```

(define (top/left->bot/right-sorted? t0)
  ;; t-wl is (listof Tree); worklist of Trees to visit
  ;;
  ;;          unvisited direct subs of visited trees
  ;; vnum is Integer; node number of most recently VISITED node
  ;; ** visited means in the dynamic flow of the tail recursion **
  ;; ** not in the static structure of the tree **
  (local [(define (fn-for-t t t-wl vnum)
    (local [(define number (node-number t)) ;unpack the fields
      (define subs (node-subs t))] ;for convenience
      (if (> number vnum)
        (fn-for-lot (append subs t-wl) number)
        false)))

    (define (fn-for-lot t-wl vnum)
      (cond [(empty? t-wl) true]
            [else
             (fn-for-t (first t-wl) (rest t-wl) vnum)])))

    (fn-for-t t0 empty (sub1 (node-number t0))))))

```

tree worklist (tree-wl)
at fn-for-t and fn-for-lot

tail recursion with worklist
calls

