Accumulators – 2 goals, 3 kinds

Goal	Kind of invariant
Preserve context from prior recursive calls	Context preserving parent house in same house…
Achieve tail recursion	Result so far rsf in sum, product Work list (for TR on tree or graph) upper left fringe of unvisited tree

Overview

- The next four lectures
  - forms of data: trees and graphs
  - recursion: structural (non-tail) and tail
  - accumulators
    - rsf
    - path in data: path, depth…
    - path in tail recursion: visited, count…
    - worklist
    - tandem worklist

l19

```
;; 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
```

```
(define (rev lox0)


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

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

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)))
```

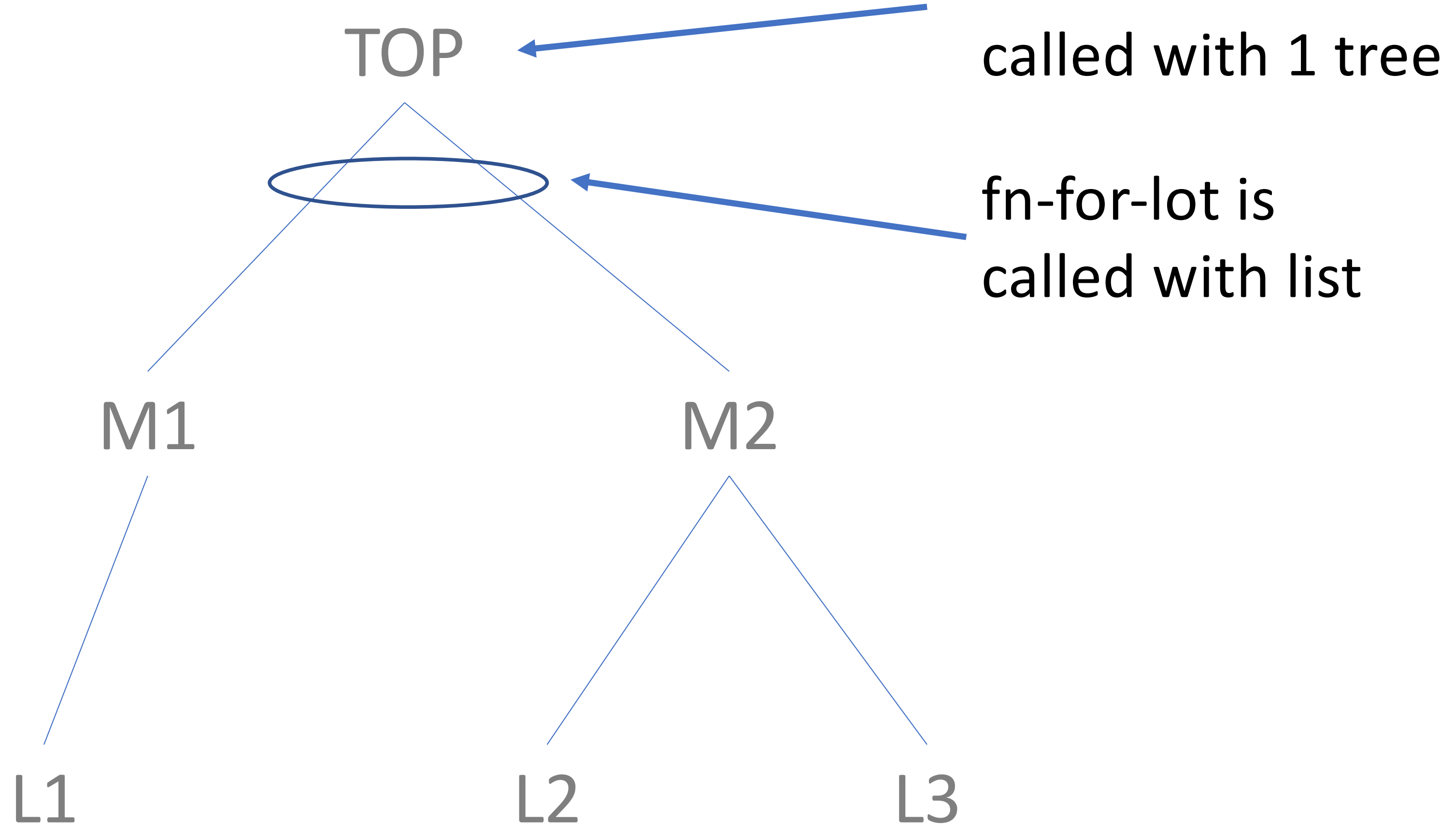
```
(define (rev lox0)
  ;; rsf is (listof X)
  ;; all elements of lox0 before (first lox), in reverse order
  (local [(define (fn-for-lox lox rsf)
            (cond [(empty? lox) rsf]
                  [else
                   (fn-for-lox (rest lox) (cons (first lox) rsf))]))]

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

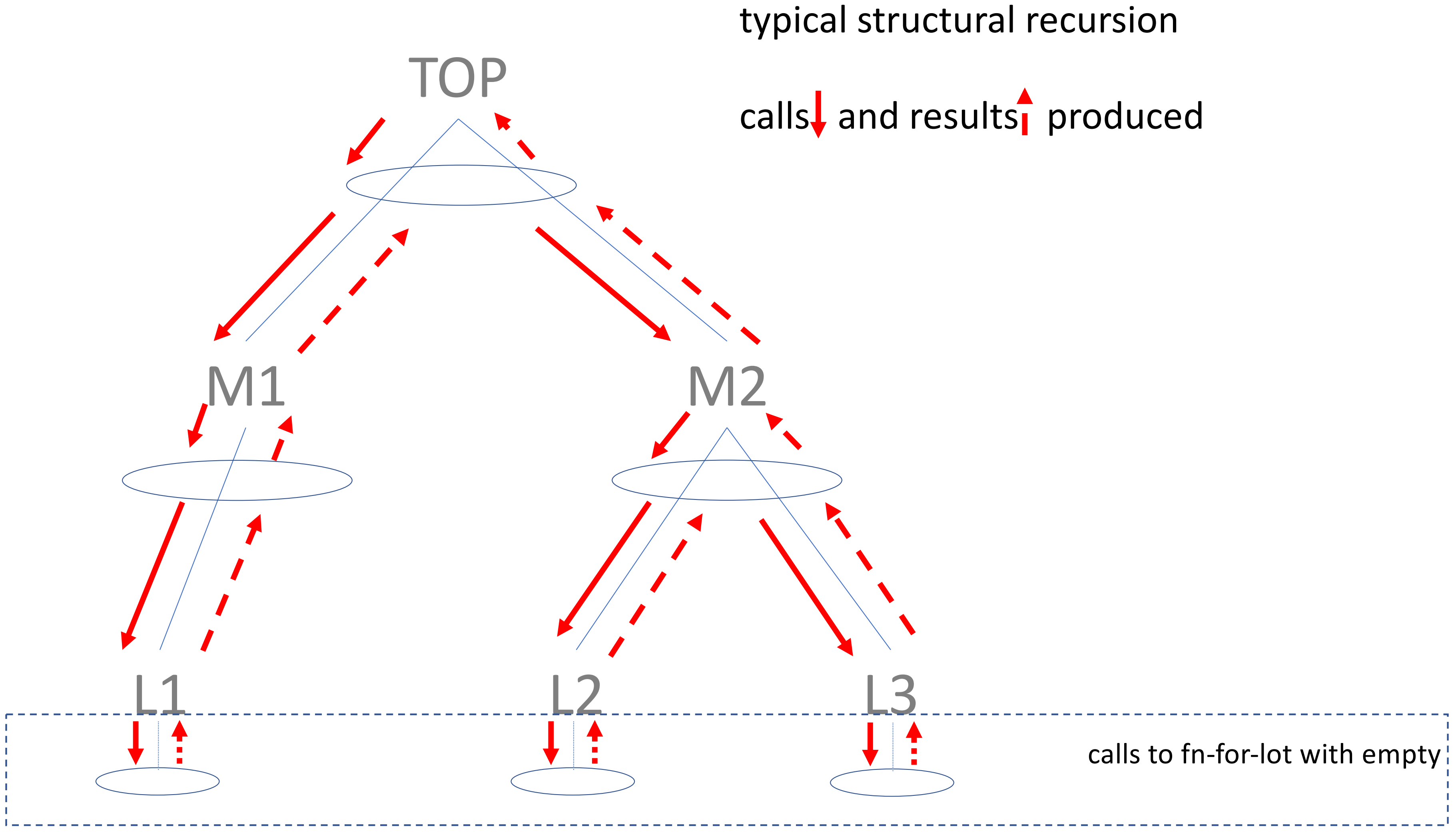
```
(@template-origin encapsulated Tree (listof Tree) try-catch)

(define (find-tree t tn)
  (local [(define (fn-for-t t)
            (local [(define name (node-name t))  ;unpack the fields
                    (define subs (node-subs t))] ;for convenience
              (if (string=? name tn)
                  t
                  (fn-for-lot subs))))

          (define (fn-for-lot lot)
            (cond [(empty? lot) false]
                  [else
                   (local [(define try (fn-for-t (first lot)))]
                     (if (not (false? try))
                         try
                         (fn-for-lot (rest lot))))]))]

    (fn-for-t t)))
```

```
(@template-origin encapsulated Tree (listof Tree) accumulator);note no try-catch

;; Tail recursion
(define (find-tree/tr t tn)
  ;; t-wl is (listof Tree)
  ;; worklist of pending trees to visit
  ;; the unvisited direct subs of all the visited trees
  ;; aka the upper left fringe of the unvisited part of original tree
  (local [(define (fn-for-t t t-wl)
            (local [(define name (node-name t))  ;unpack the fields
                    (define subs (node-subs t))] ;for convenience
              (if (string=? name tn)
                  t
                  (fn-for-lot (append subs t-wl)))))

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

    (fn-for-t t empty)))
```

```
(@template-origin encapsulated Tree (listof Tree) try-catch)

(define (find-tree t tn)
  (local [(define (fn-for-t t)
            (local [(define name (node-name t))  ;unpack the fields
                    (define subs (node-subs t))] ;for convenience
              (if (string=? name tn)
                  t
                  (fn-for-lot subs))))

          (define (fn-for-lot lot)
            (cond [(empty? lot) false]
                  [else
                   (local [(define try (fn-for-t (first lot)))]
                     (if (not (false? try))
                         try
                         (fn-for-lot (rest lot))))]))]

    (fn-for-t t)))
```

NOTATION:

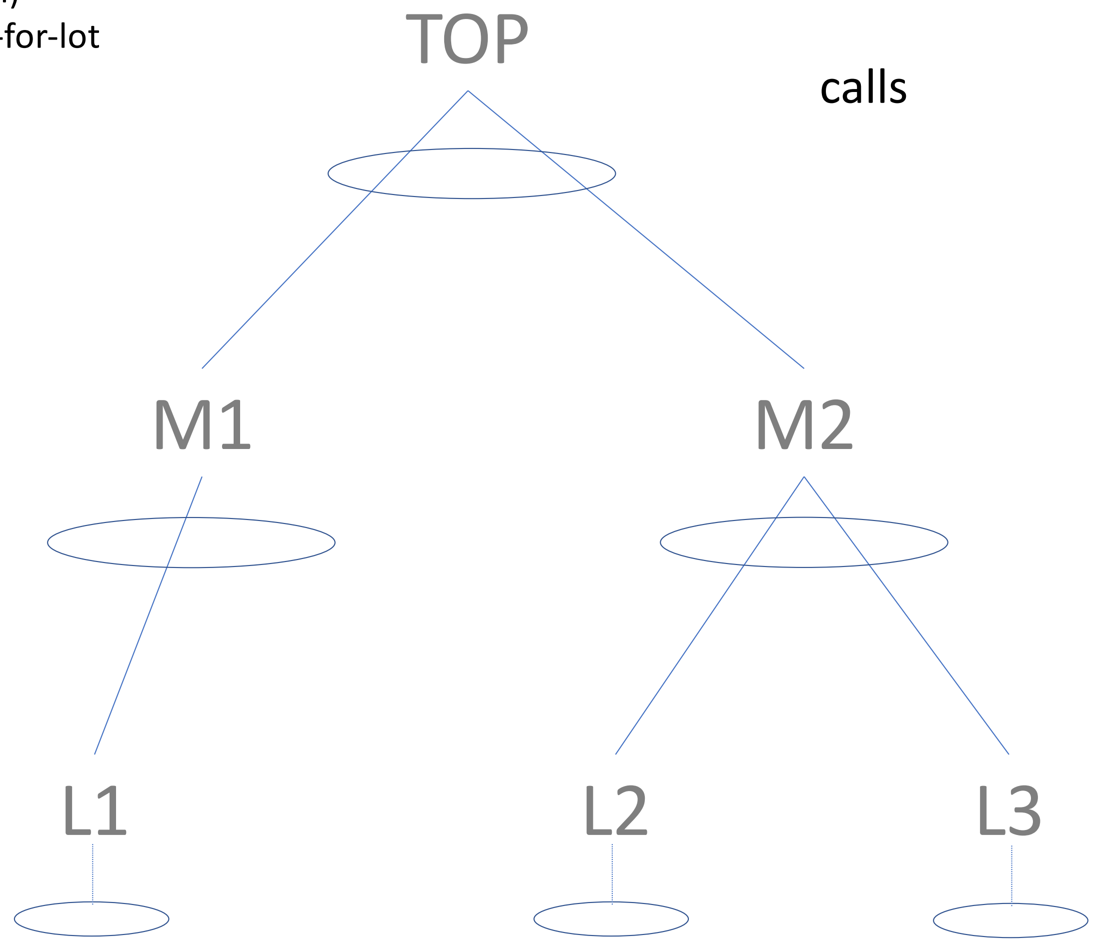

TOP
M1
M2
L1
L2
L3
fn-for-t is
called with 1 tree
fn-for-lot is
called with list


typical structural recursion
calls and results produced
TOP
M1
M2
L1
L2
L3
calls to fn-for-lot with empty

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

tail recursion with worklist

calls



```
(@template-origin encapsulated Tree (listof Tree) try-catch)

(define (find-tree t tn)
  (local [(define (fn-for-t t)
            (local [(define name (node-name t))  ;unpack the fields
                    (define subs (node-subs t))] ;for convenience
              (if (string=? name tn)
                  t
                  (fn-for-lot subs))))

          (define (fn-for-lot lot)
            (cond [(empty? lot) false]
                  [else
                   (local [(define try (fn-for-t (first lot)))]
                     (if (not (false? try))
                         try
                         (fn-for-lot (rest lot))))]))]

    (fn-for-t t)))
```

l20

Overview

- The next three lectures
  - forms of data: trees and graphs
  - recursion: structural and tail
  - accumulators
    - rsf (path)
    - path in data: path, depth…
    - path in tail recursion: visited, count…
    - worklist
    - tandem worklist l20

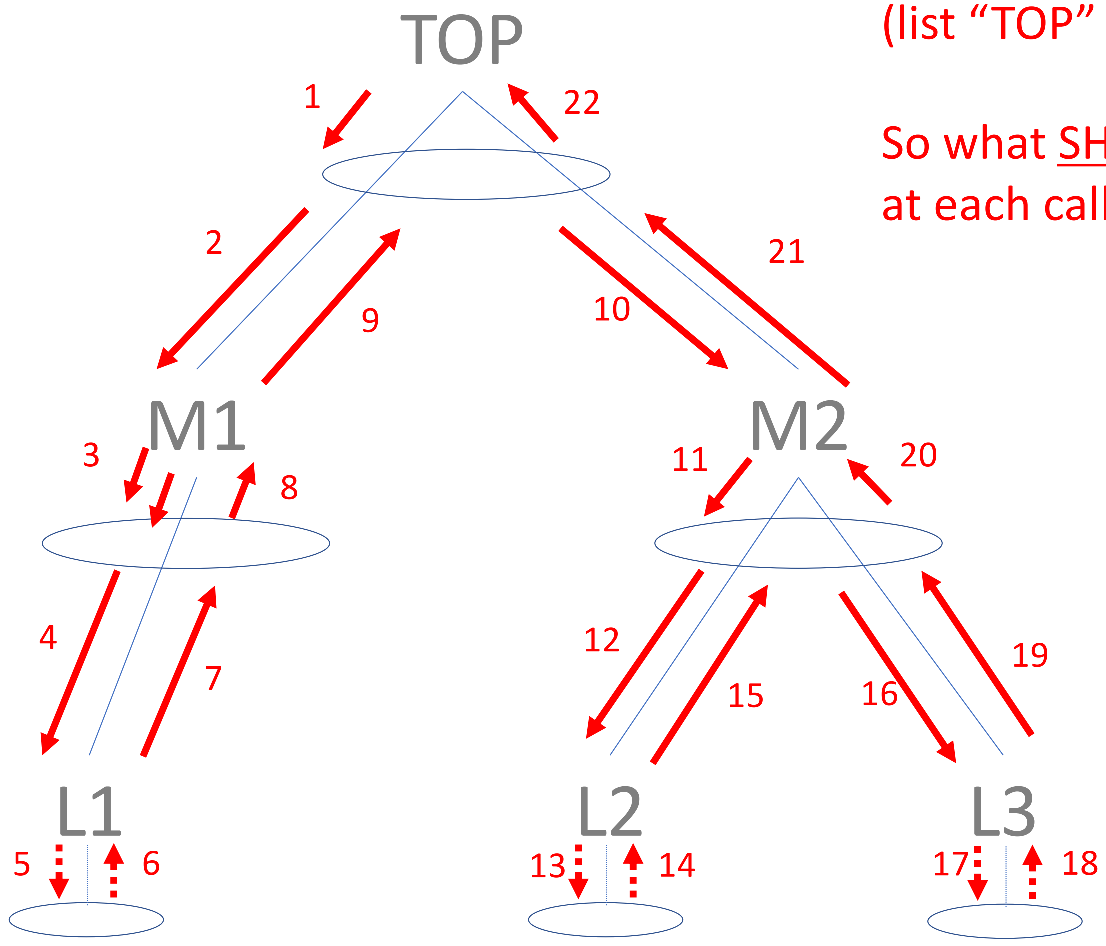
Accumulators – 2 goals, 3 kinds

Goal	Kind of invariant
Preserve context from prior recursive calls	Context preserving parent house in same house…
Achieve tail recursion	Result so far rsf in sum, product Work list (for TR on tree or graph) upper left fringe of unvisited tree

structural recursion

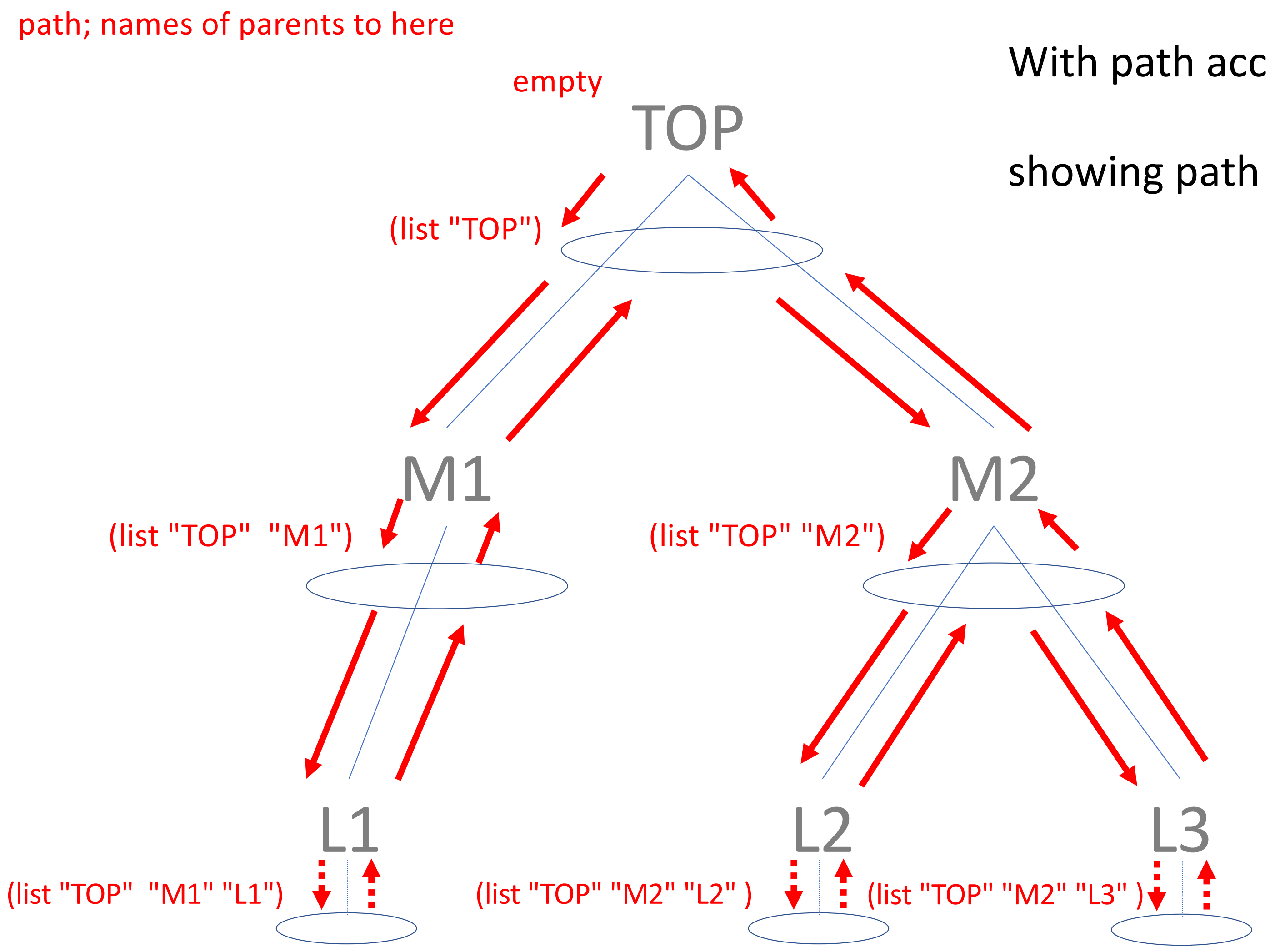
(find-path TOP "L2") must produce
(list "TOP" "M2" "L2")

So what SHOULD BE value of path
at each call to fn-for-lot?



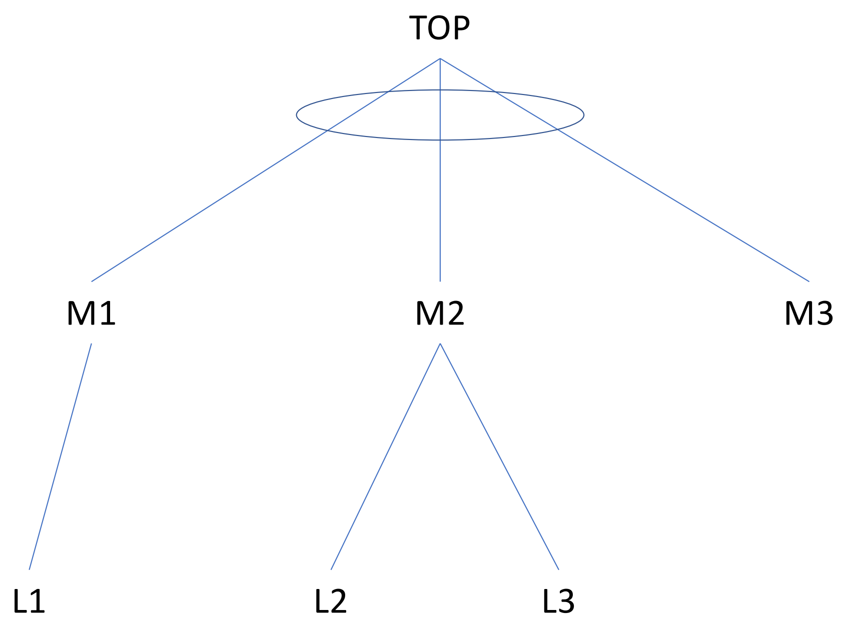
With path accumulator

showing path at each call to fn-for-lot



[3 minutes]
Given the tree to the right,and given a TAIL-RECURSIVE traversal of the tree, as in find-tree last time, consider the call to fn-for-t with a first argument of L2. What will be the value of t-wl at that call?

(A) (fn-for-t L2 (list M3))
(B) (fn-for-t L2 (list TOP M1 L1 M2))
(C) (fn-for-t L2 (list L2 M3))
(D) (fn-for-t L2 (list L3 M3))
(E) (fn-for-t L2 (list L2 L3 M3))

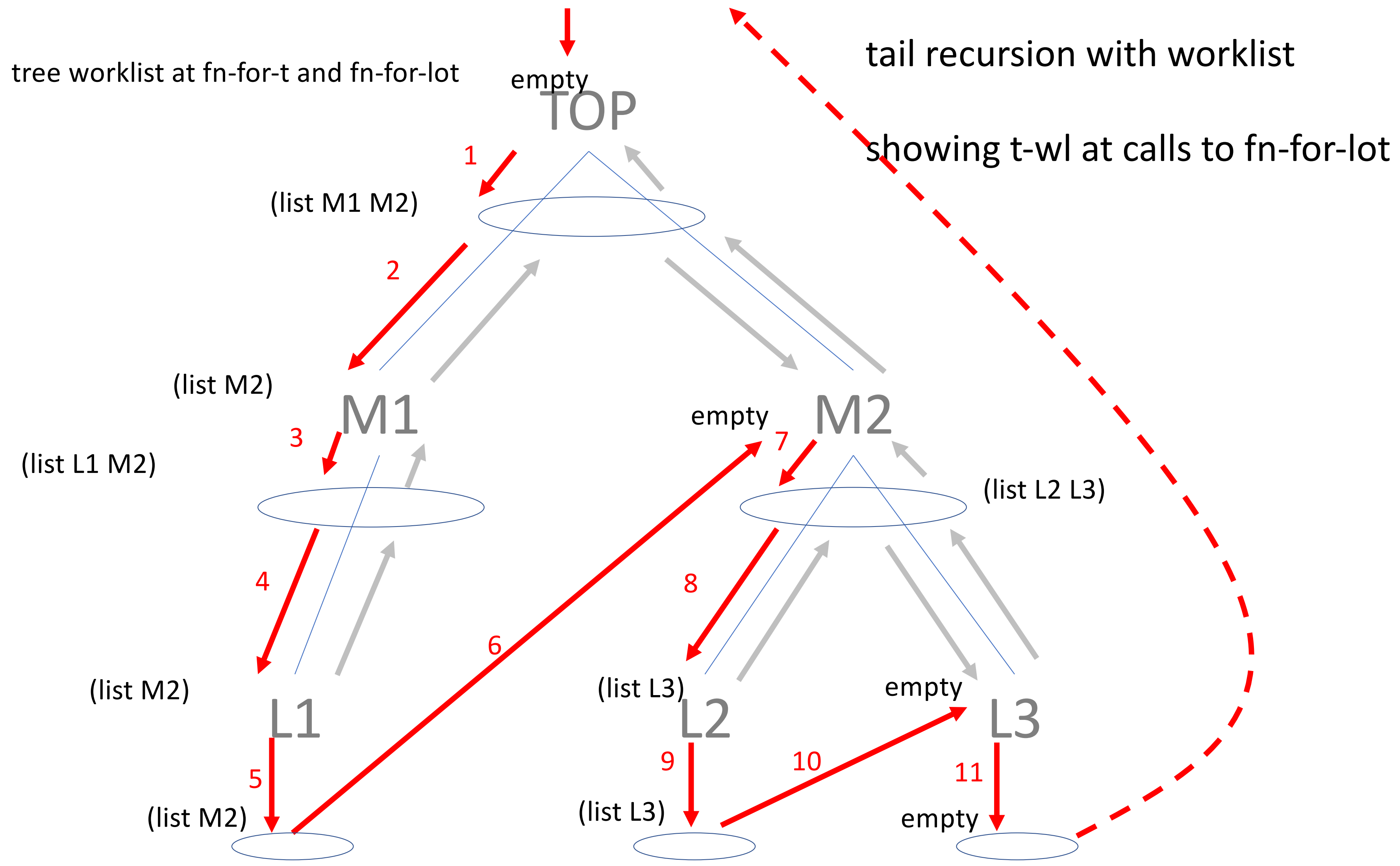


worklist is unvisited direct subs of visited nodes

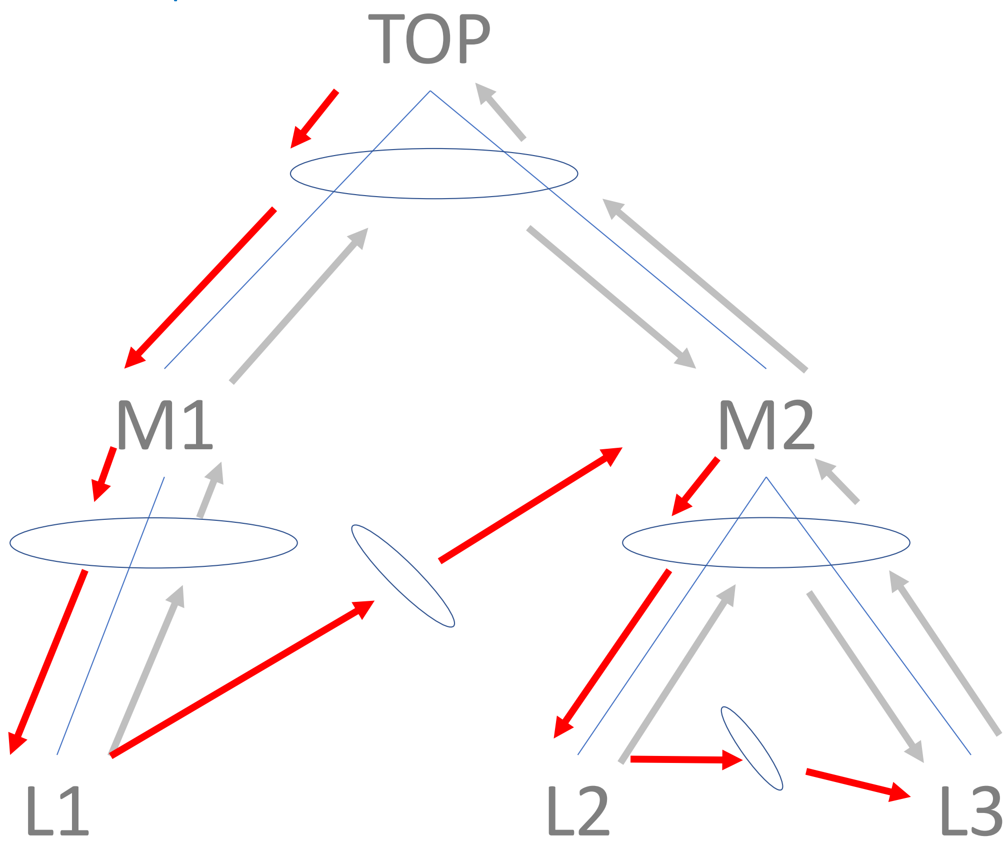
```
(@template-origin encapsulated Tree (listof Tree) accumulator);note no try-catch

;; Tail recursion
(define (find-tree/tr t tn)
  ;; t-wl is (listof Tree)
  ;; worklist of pending trees to visit
  ;; the unvisited direct subs of all the visited trees
  ;; aka the upper left fringe of the unvisited part of original tree
  (local [(define (fn-for-t t t-wl)
            (local [(define name (node-name t))  ;unpack the fields
                    (define subs (node-subs t))] ;for convenience
              (if (string=? name tn)
                  t
                  (fn-for-lot (append subs t-wl)))))

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

    (fn-for-t t empty)))
```

tail recursion with worklist

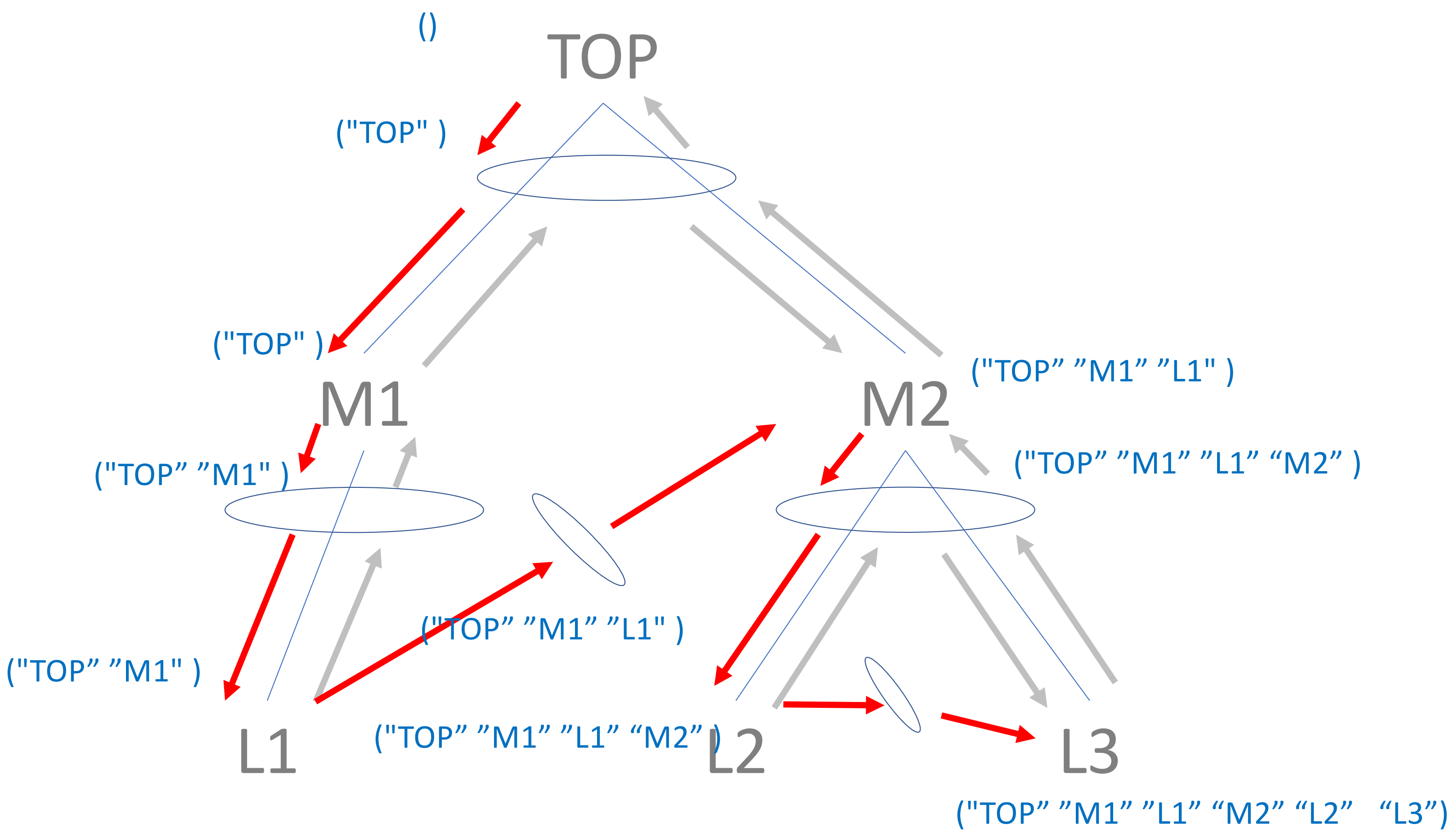
showing t-wl at calls to fn-for-lot



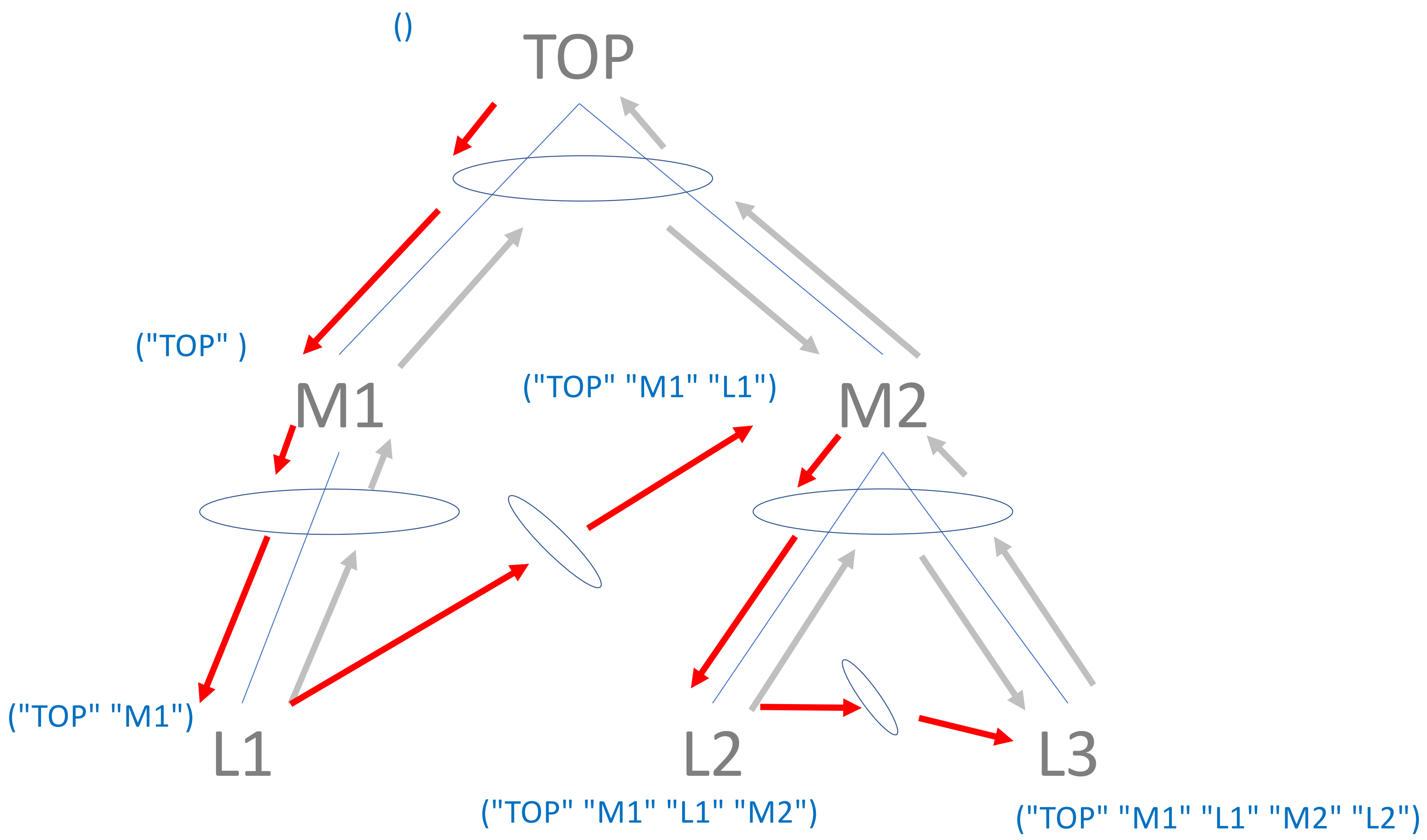
fill in ??? at calls to fn-for-t
can do fn-for-lot too if that helps

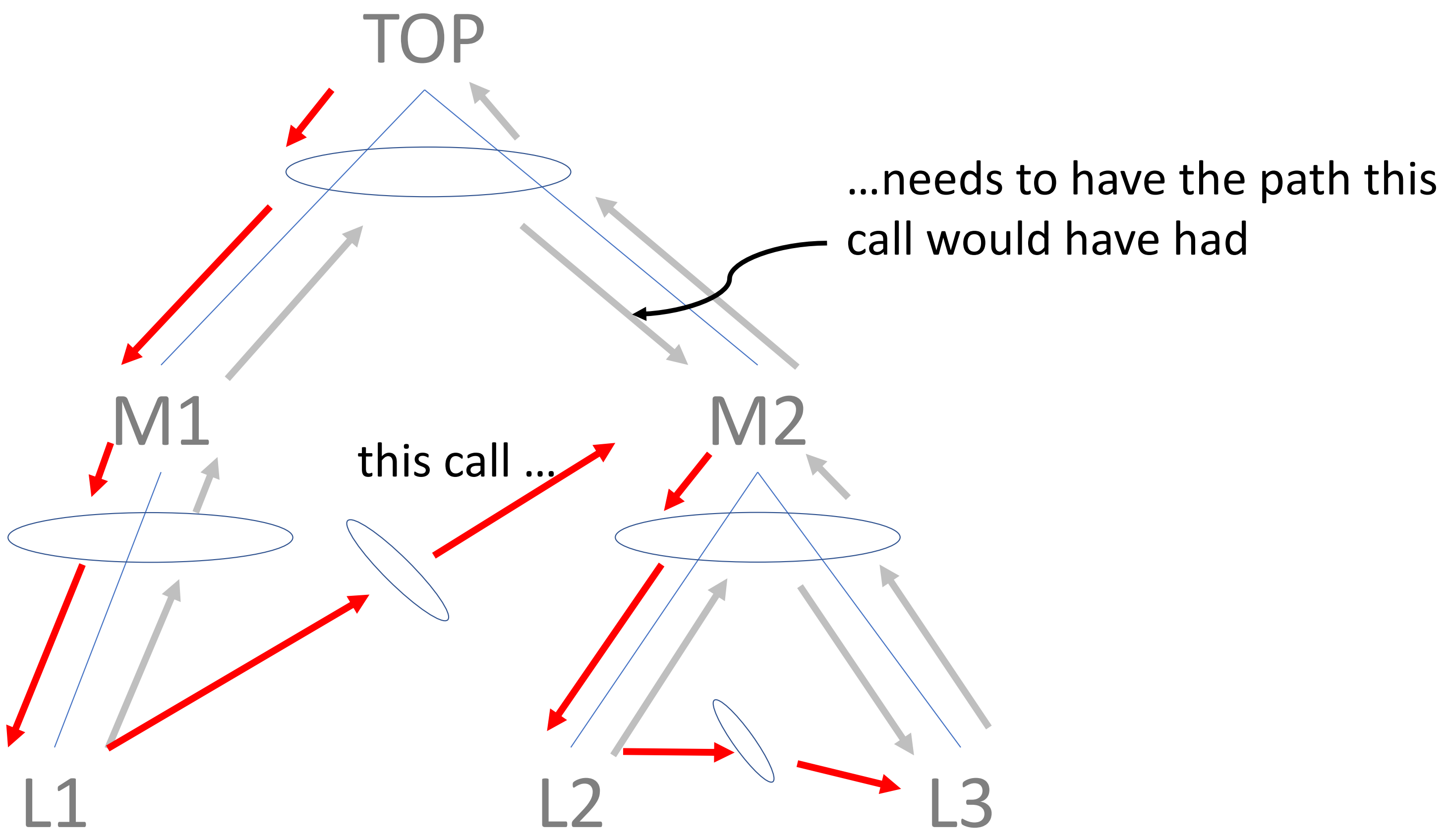


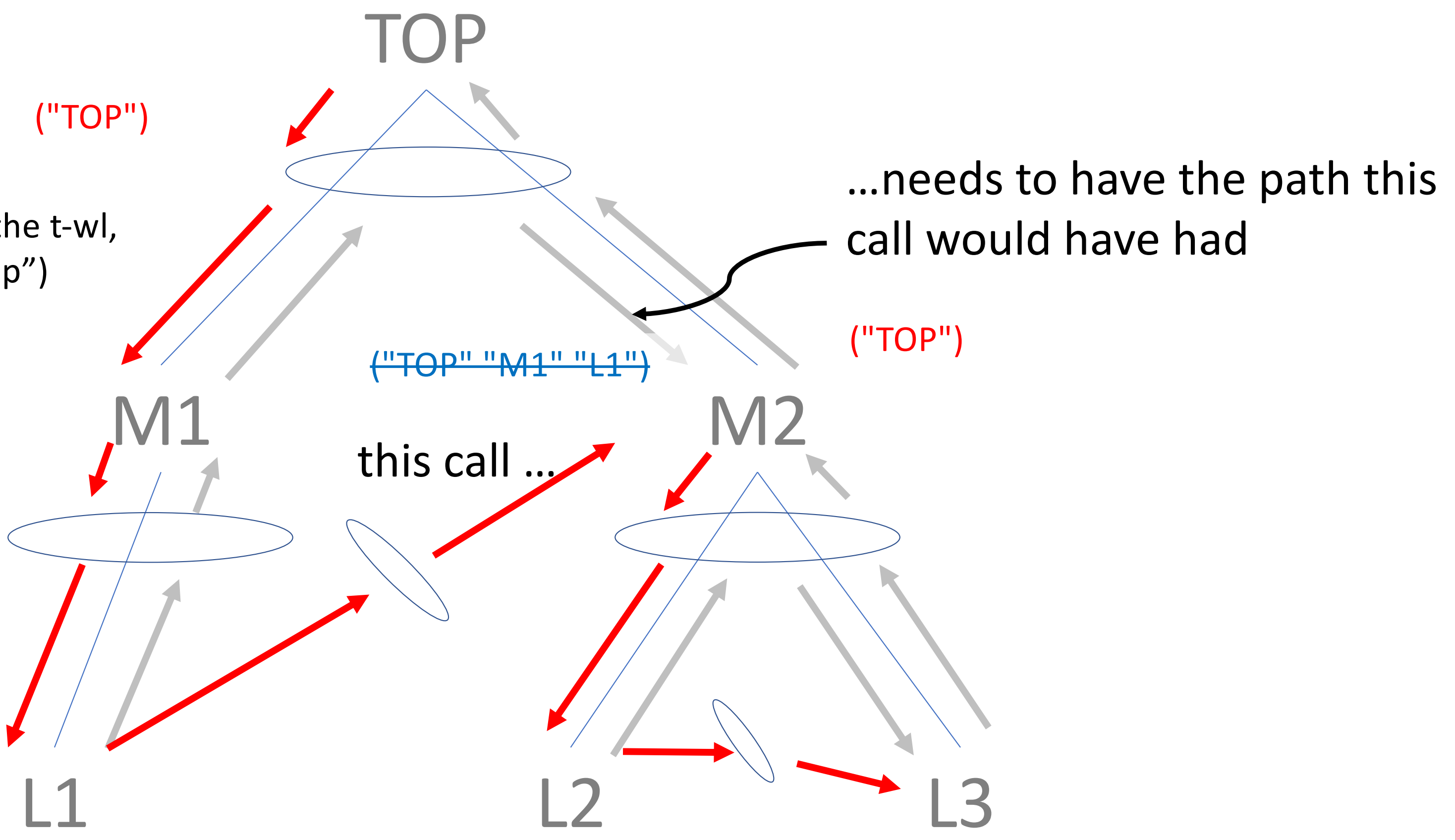
fill in visited at calls to fn-for-t



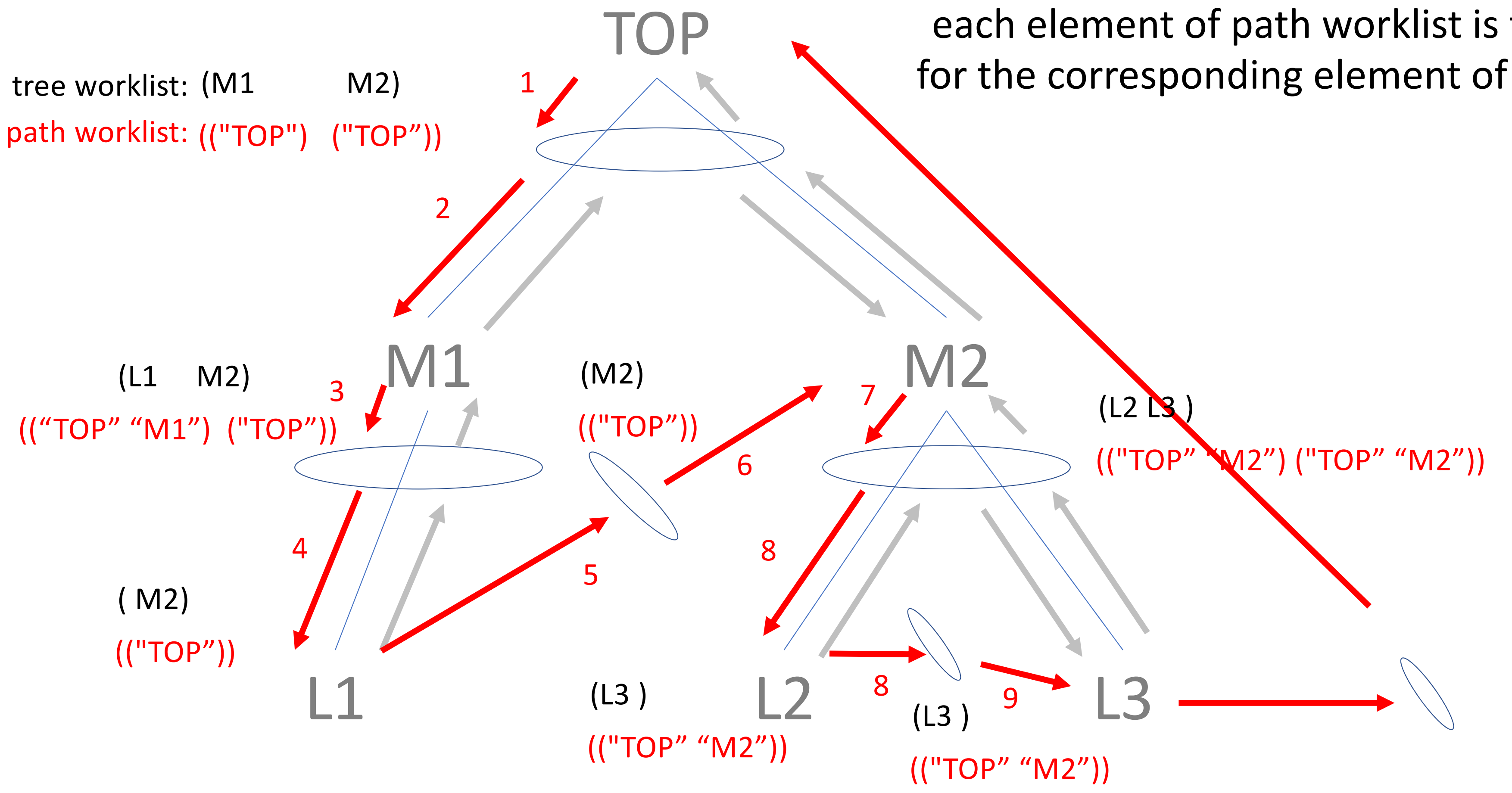
fill in visited at calls to fn-for-t




TOP
…needs to have the path this call would have had
M1
this call …
M2
L1
L2
L3

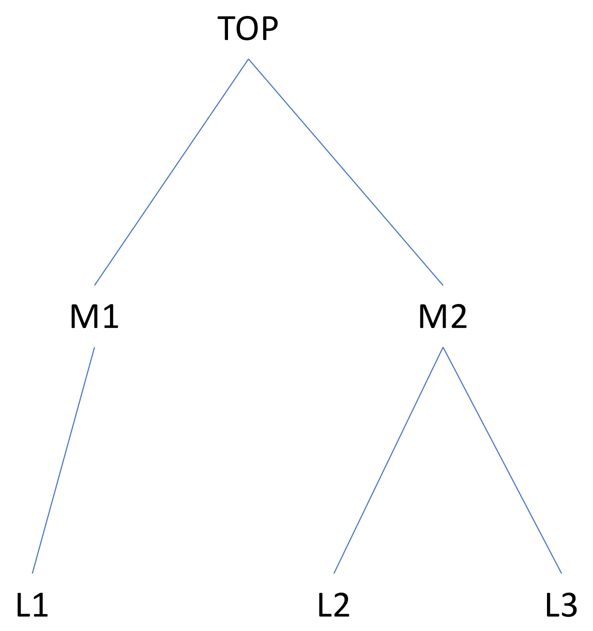

TOP
("TOP")
when we put M2 on the t-wl,
we had a path of ("Top")
…needs to have the path this
call would have had
~~("TOP" "M1" "L1")~~
("TOP")
M1
M2
this call …
L1
L2
L3

tail recursion with TANDEM WORKLISTS
each element of path worklist is the path
for the corresponding element of the tree
worklist



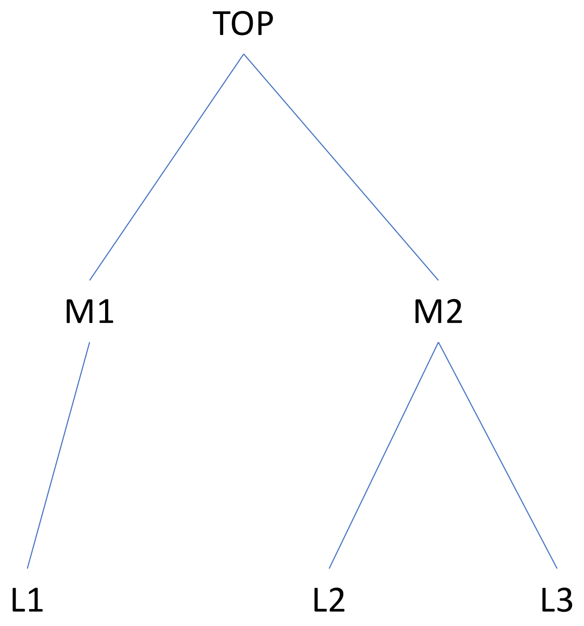
arb arity tree structural recursion templates

```
(@template Tree (listof Tree) encapsulated)

(define (fn-for-tree t)


  (local [(define (fn-for-t t)
            (local [(define name (node-name t))  ;unpack the fields
                    (define subs (node-subs t))] ;for convenience

              (... name (fn-for-lot subs))))



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

    (fn-for-t t)))
```



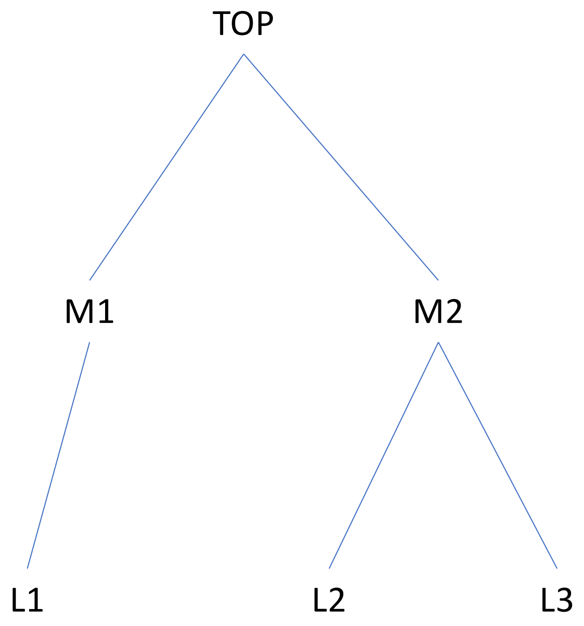
find-path: structural recursion with path accumulator.

```
(@template Tree (listof Tree) accumulator)

(define (find-path t n)
  ;; path is (listof String); names of ... grandparent, parent trees
  ;;                          (builds along recursive  calls)
  (local [(define (fn-for-t t path)
            (local [(define name (node-name t))
                    (define subs (node-subs t))
                    (define npath (append path (list name)))]
              (if (string=? name n)
                  npath
                  (fn-for-lot subs npath))))

          (define (fn-for-lot lot path)
            (cond [(empty? lot) false]
                  [else
                   (local [(define try (fn-for-t (first lot) path))]
                     (if (not (false? try))
                         try
                         (fn-for-lot (rest lot) path)))]))]


    (fn-for-t t empty)))
```



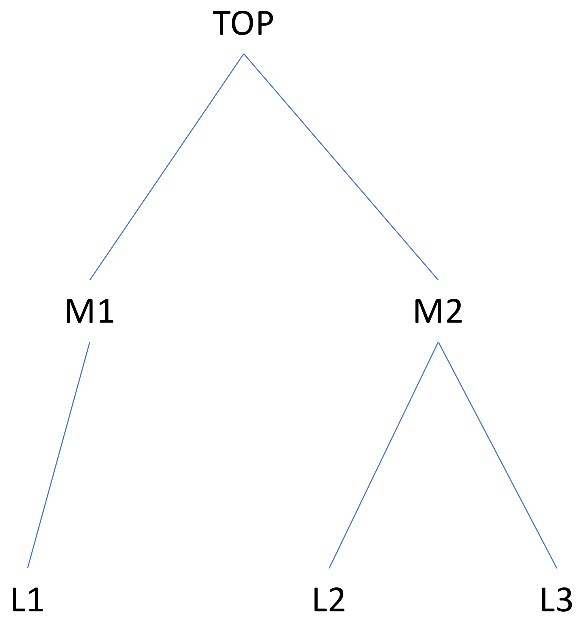
find-tree: tail recursion with worklist

```
(@template backtracking Tree (listof Tree) accumulator)

(define (find-tree t to)
  ;; t-wl is (listof Tree)
  ;; worklist of pending trees to visit
  (local [(define (fn-for-t t t-wl)
            (local [(define name (node-name t))  ;unpack the fields
                    (define subs (node-subs t))] ;for convenience
              (if (string=? name to)
                  t
                  (fn-for-lot (append subs t-wl)))))


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


    (fn-for-t t empty)))
```



tail recursion with tandem worklists (tree and path), also visited

```
(@template Tree (listof Tree) accumulator)

(define (find-path t to)
  ;; t-wl is (listof Tree)
  ;; worklist of trees to visit (unvisited subs of already visited trees)
  ;;
  ;; p-wl is (listof (listof String))
  ;; worklist of paths to corresponding trees in t-wl
  ;;
  ;; visited is (listof String)
  ;; names of trees visited so far (builds along tail recursive calls)
  (local [(define (fn-for-t t path t-wl p-wl visited)
            (local [(define name (node-name t))
                    (define subs (node-subs t))
                    (define npath (append path (list name)))
                    (define nvisited (append visited (list name)))]
              (if (string=? name to)
                  npath
                  (fn-for-lot (append                          subs  t-wl)
                              (append (map (lambda (s) npath) subs) p-wl)
                              nvisited))))

          (define (fn-for-lot t-wl p-wl visited)
            (cond [(empty? t-wl) false]
                  [else
                   (fn-for-t (first t-wl)
                             (first p-wl)
                             (rest t-wl)
                             (rest p-wl)
                             visited)]))]

    (fn-for-t t empty  empty empty empty)))
```


TOP
M1
M2
L1
L2
L3