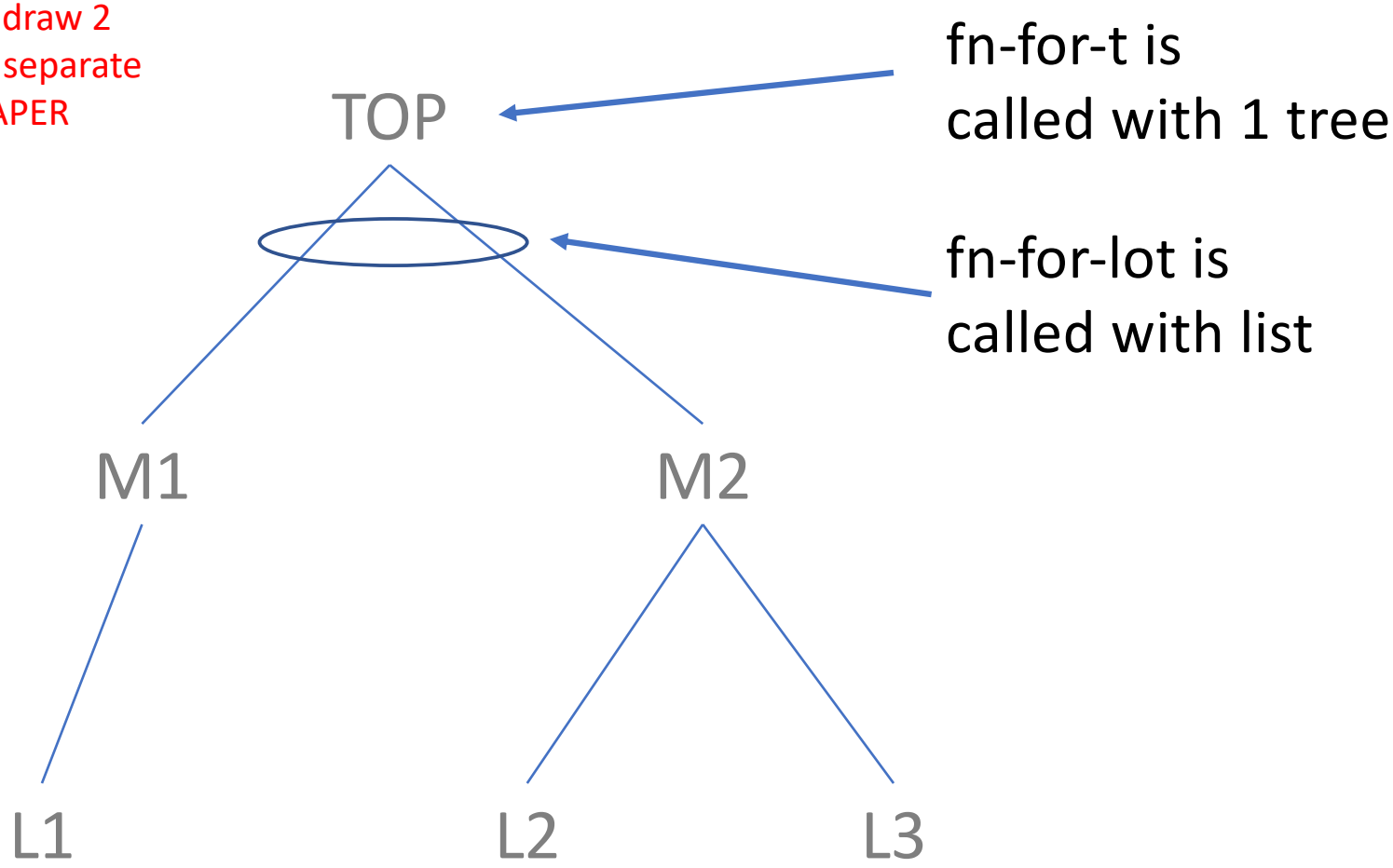
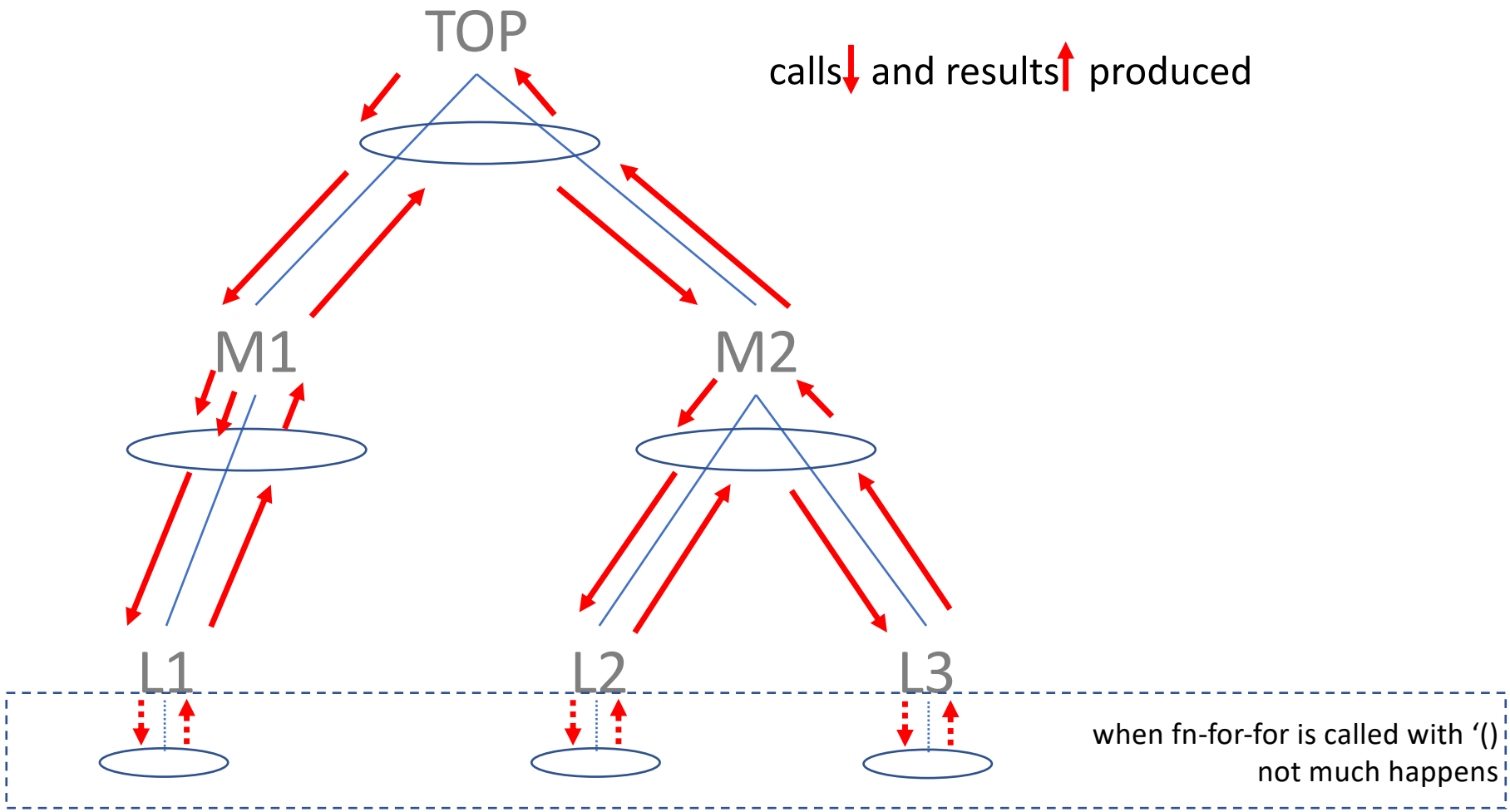
Lecture 19

While waiting, draw 2 copies of this on separate sheets on PAPER



typical structural recursion

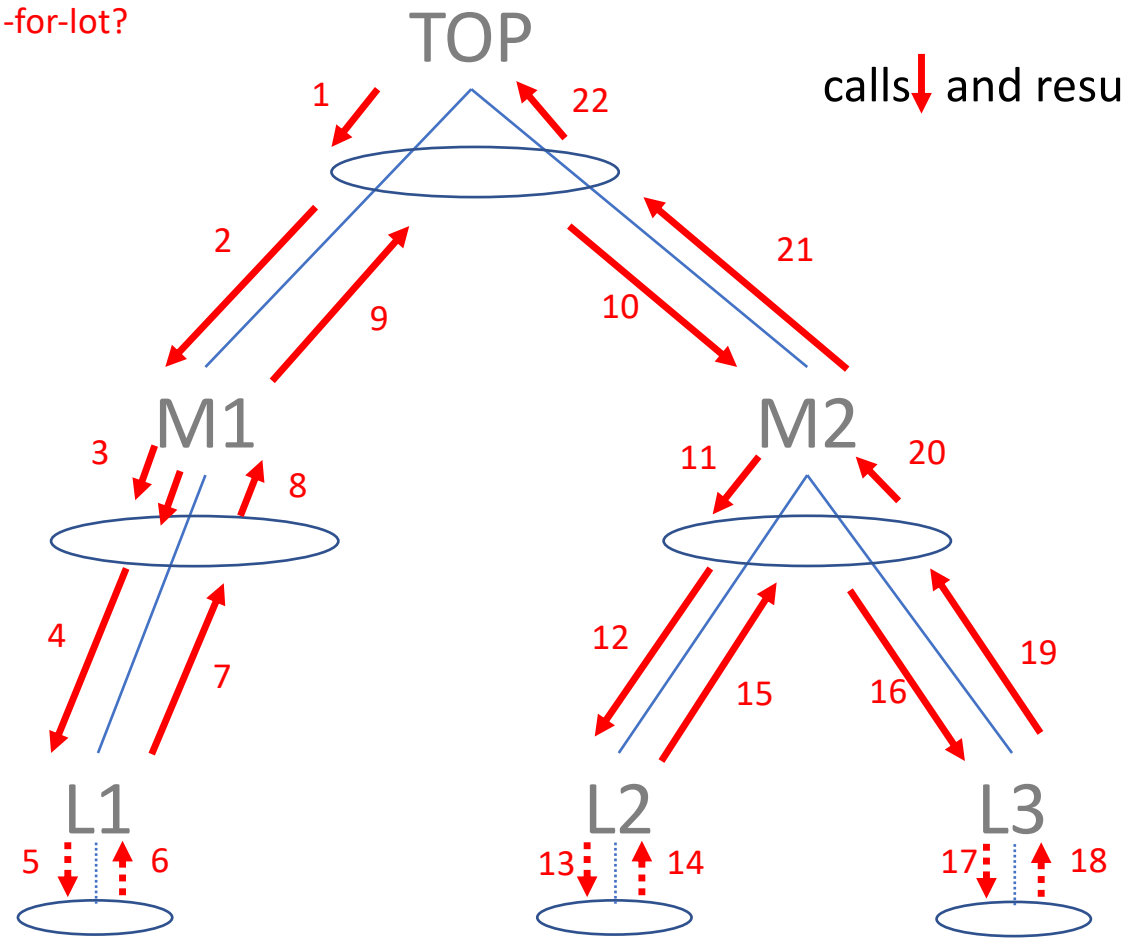
calls↓ and results↑ produced



What is value of path at each call to fn-for-lot?

typical structural recursion

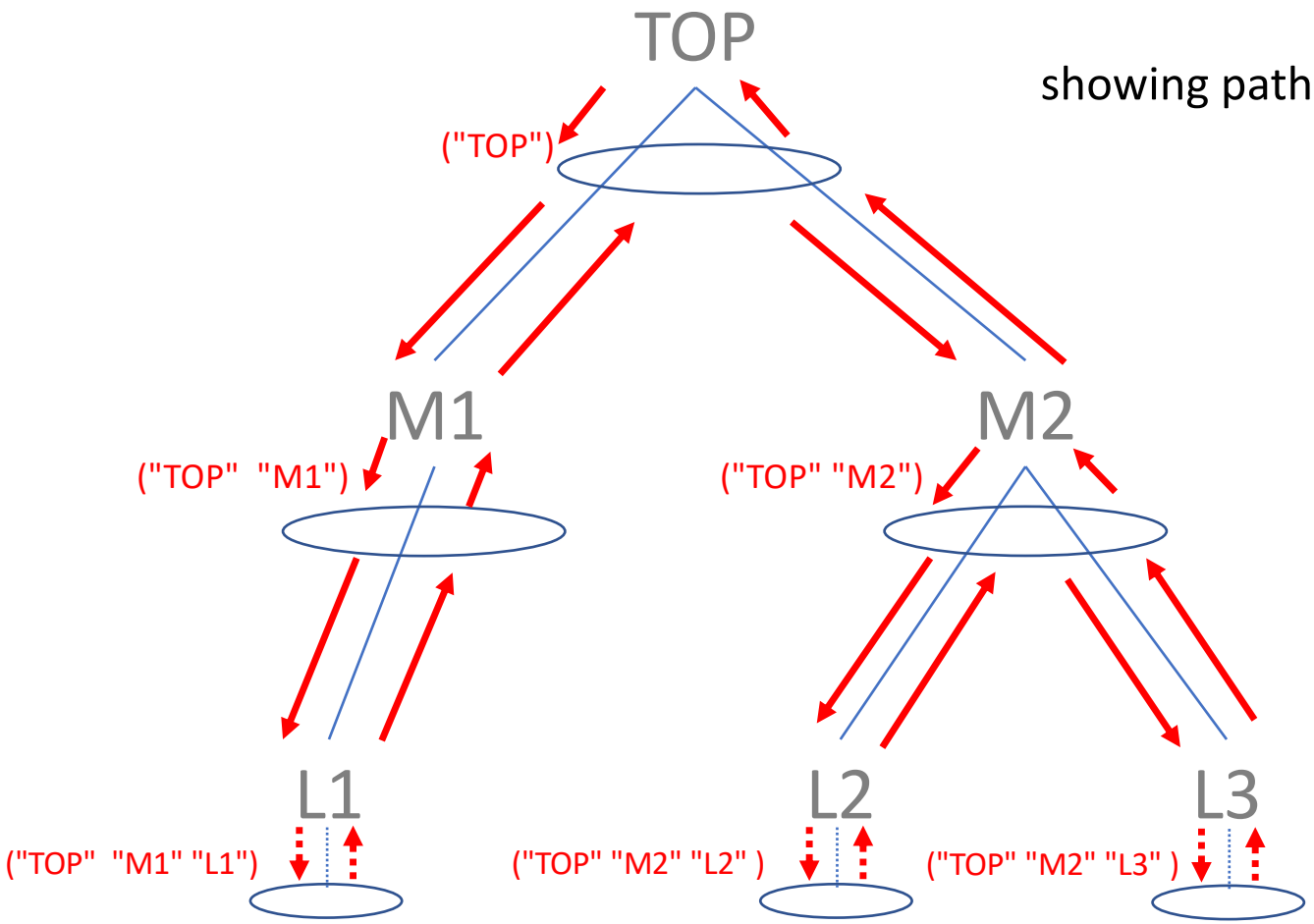
calls↓ and results↑ produced



path; names of parents to here

With path accumulator

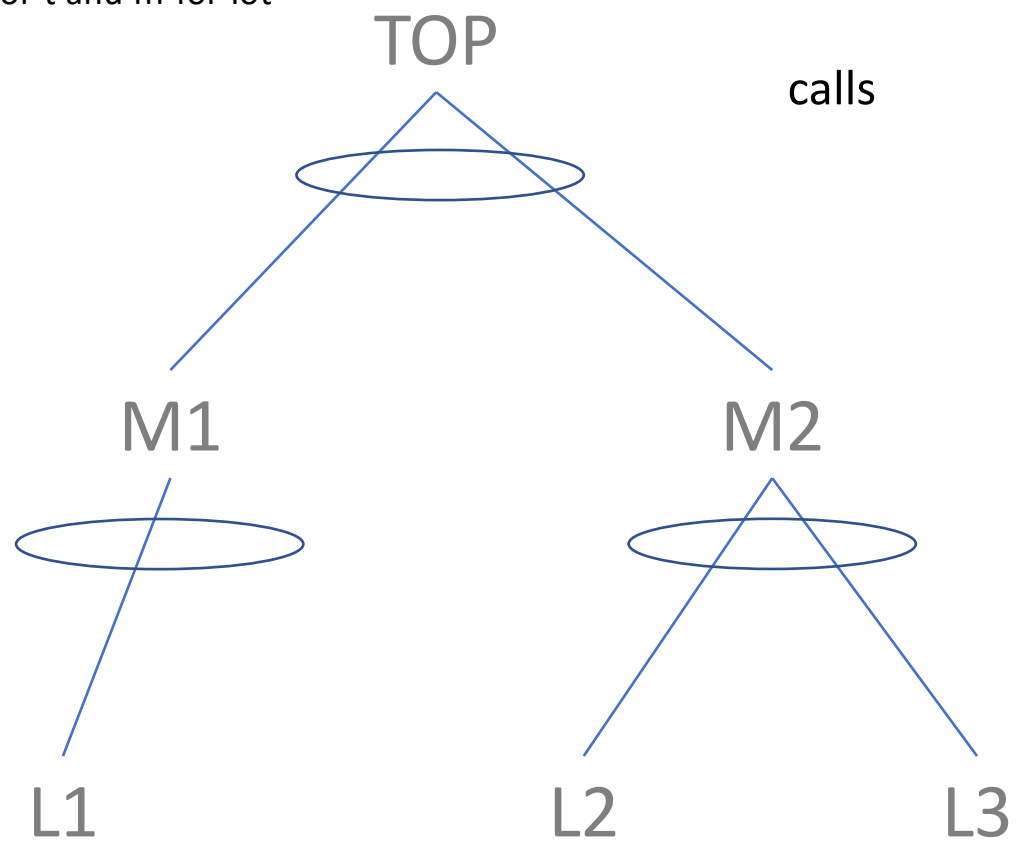
showing path at each call to fn-for-lot



tree worklist at fn-for-t and fn-for-lot

tail recursion with worklist

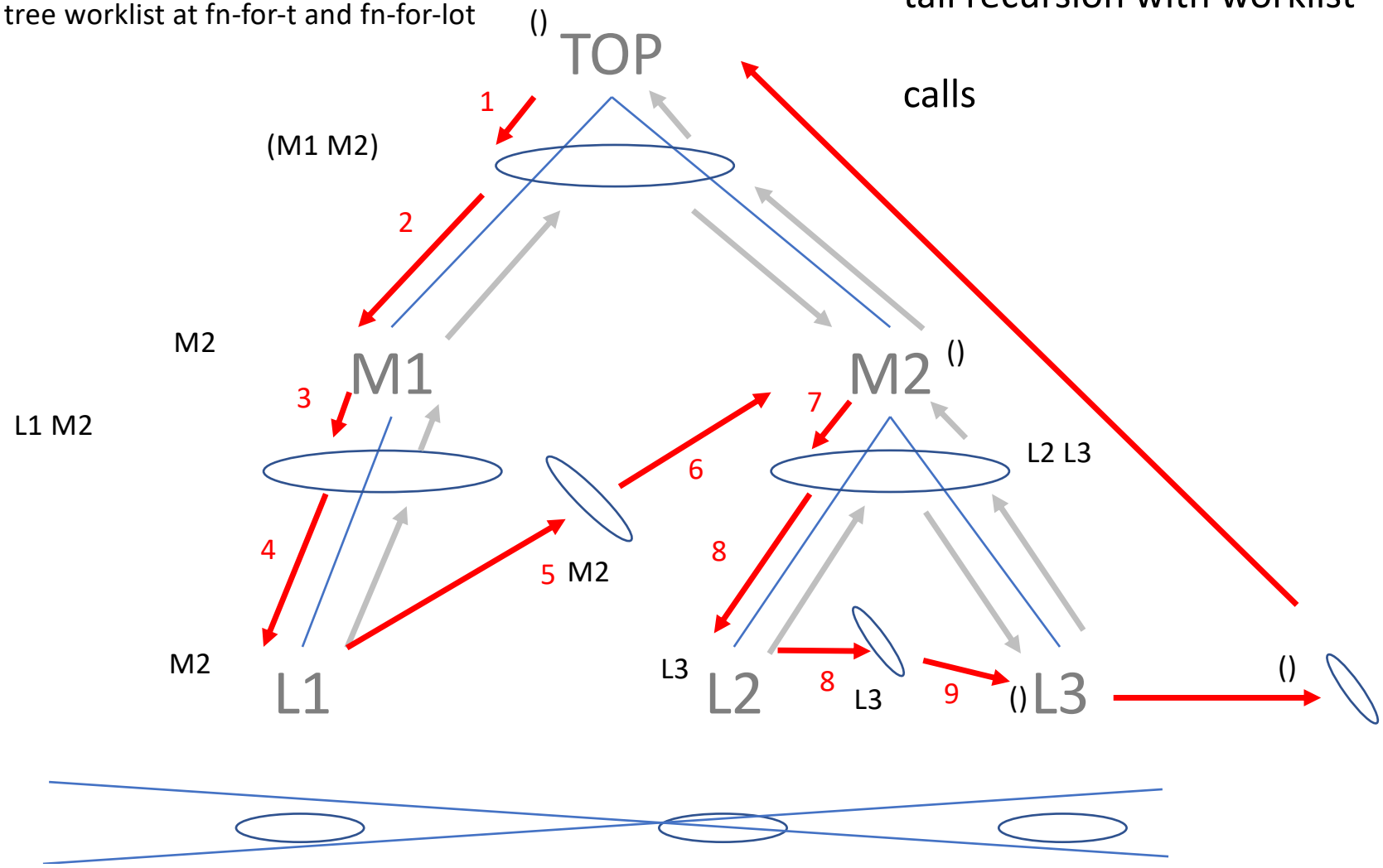
calls



tree worklist at fn-for-t and fn-for-lot

tail recursion with worklist

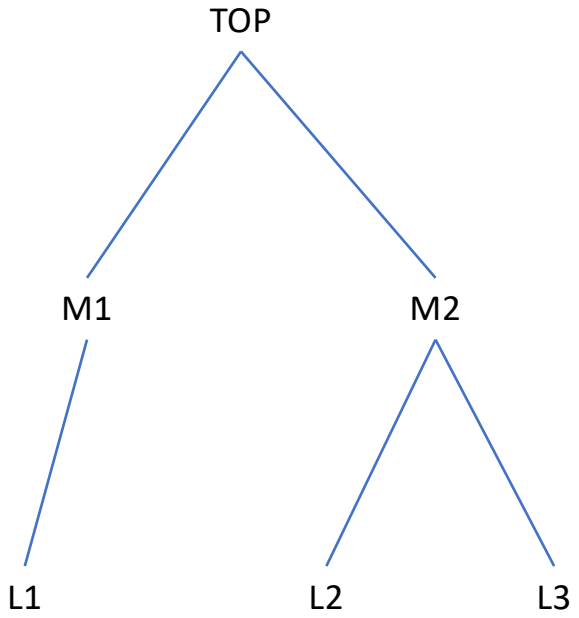
calls



Lecture 20

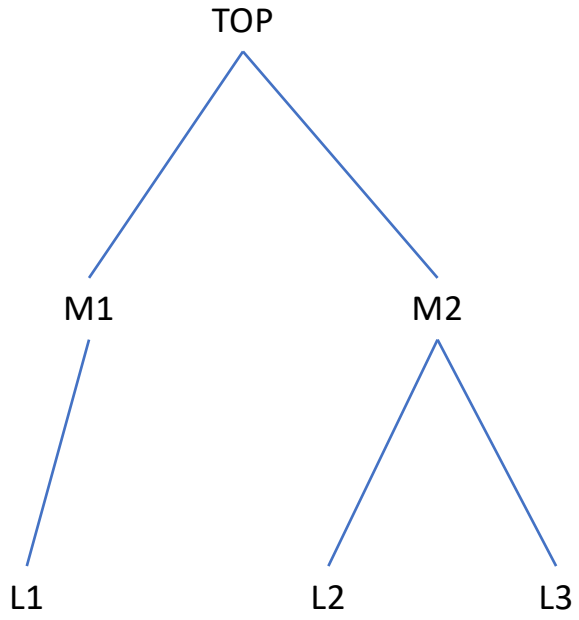
arb arity tree structural recursion templates

```
(@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)))
```



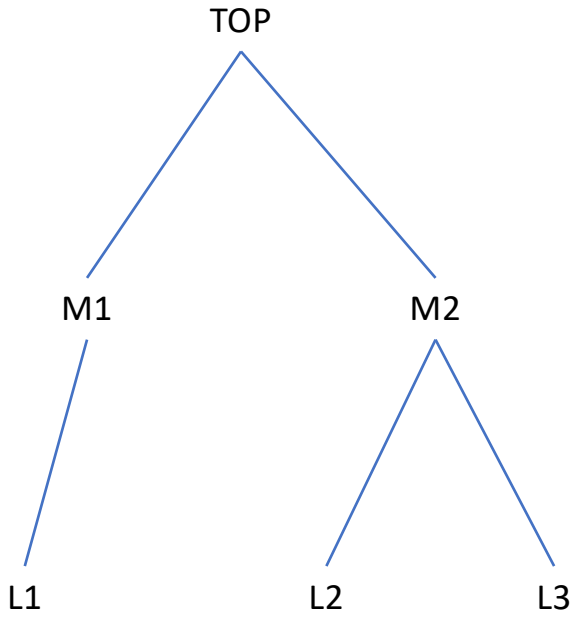
find-path: structural recursion with path accumulator.

```
(@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)))
```

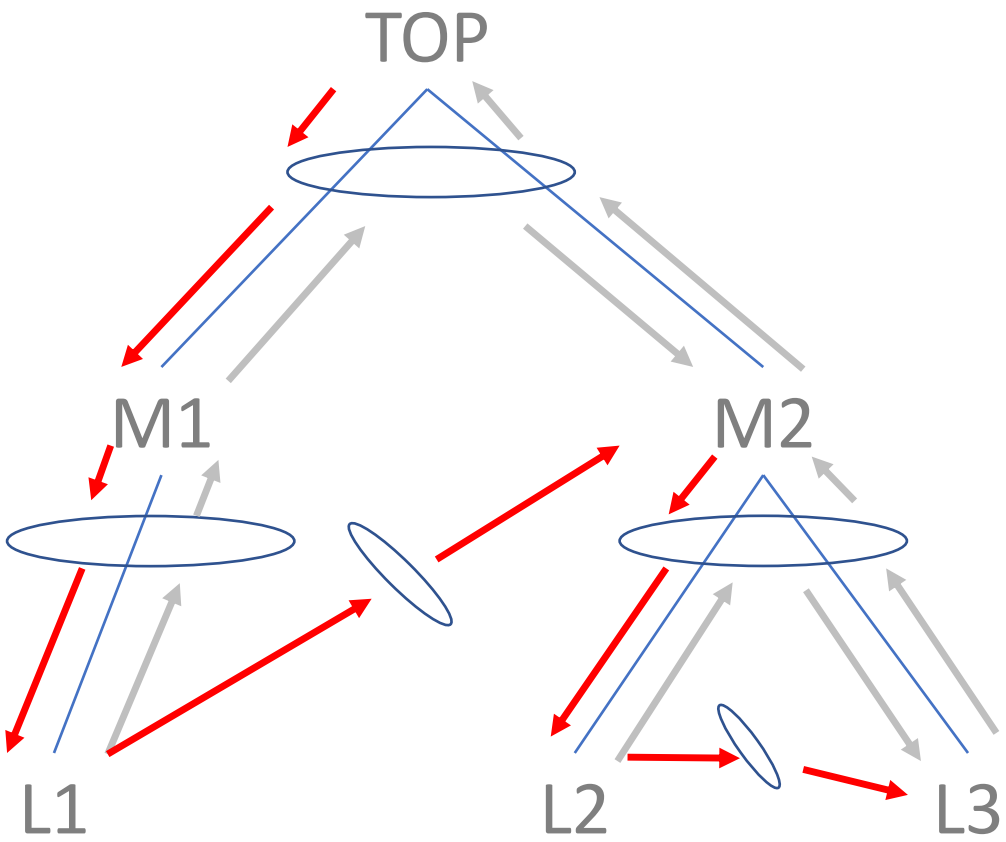


find-tree: tail recursion with worklist

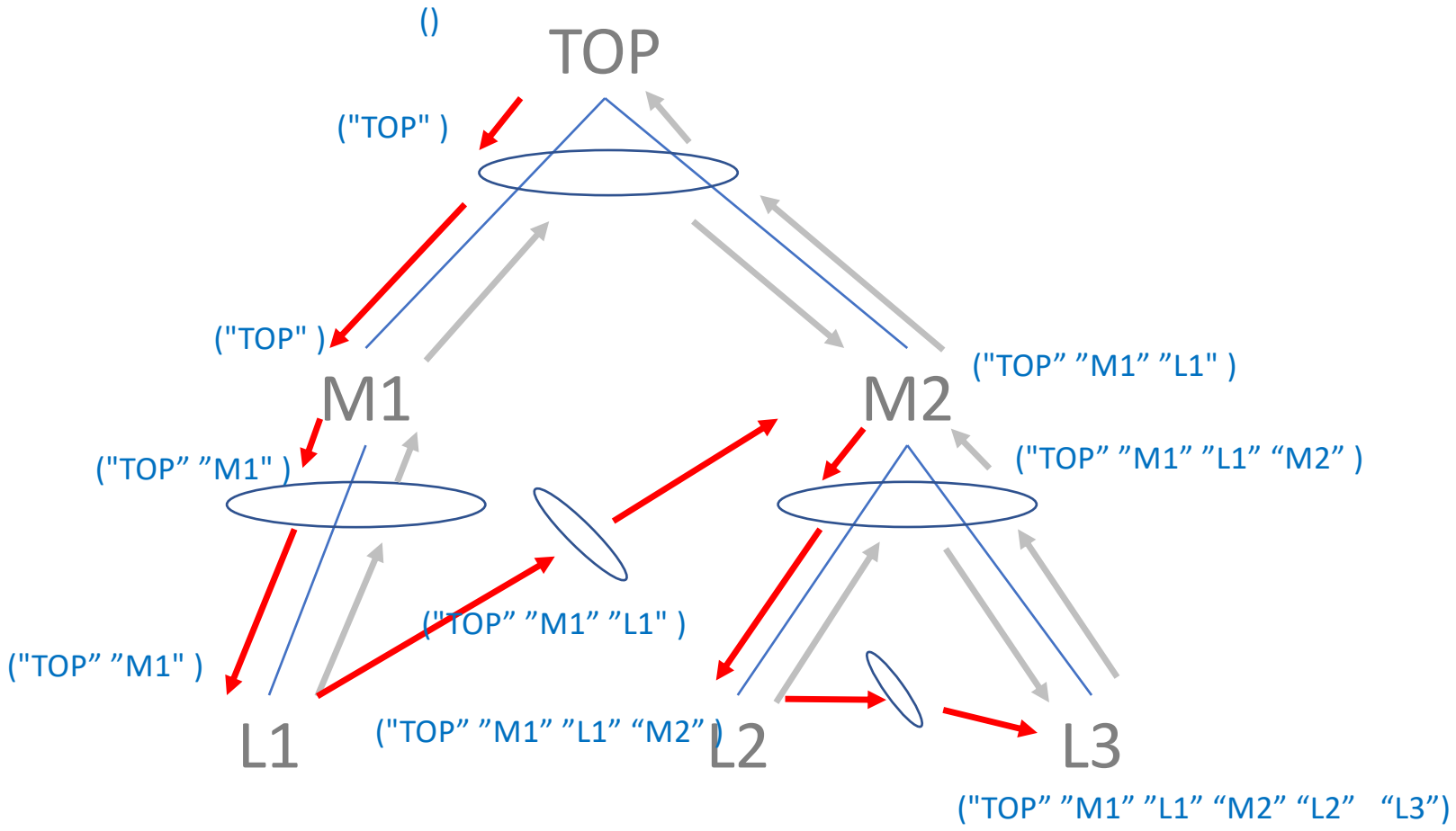
```
(@template backtracking Tree (listof Tree) accumulator)

(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)))
```



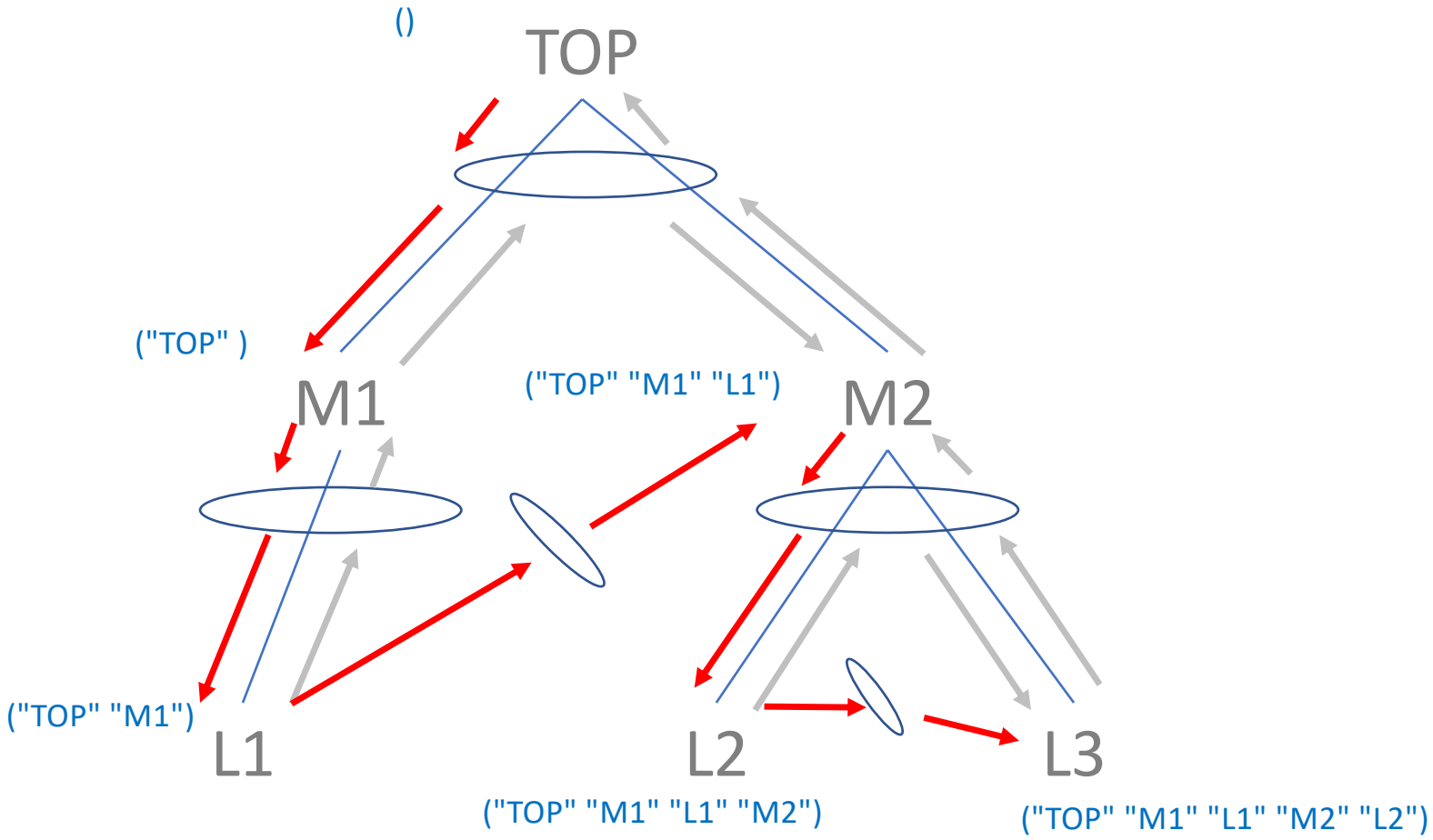
fill in visited at calls to fn-for-t
can do fn-for-lot too if that helps

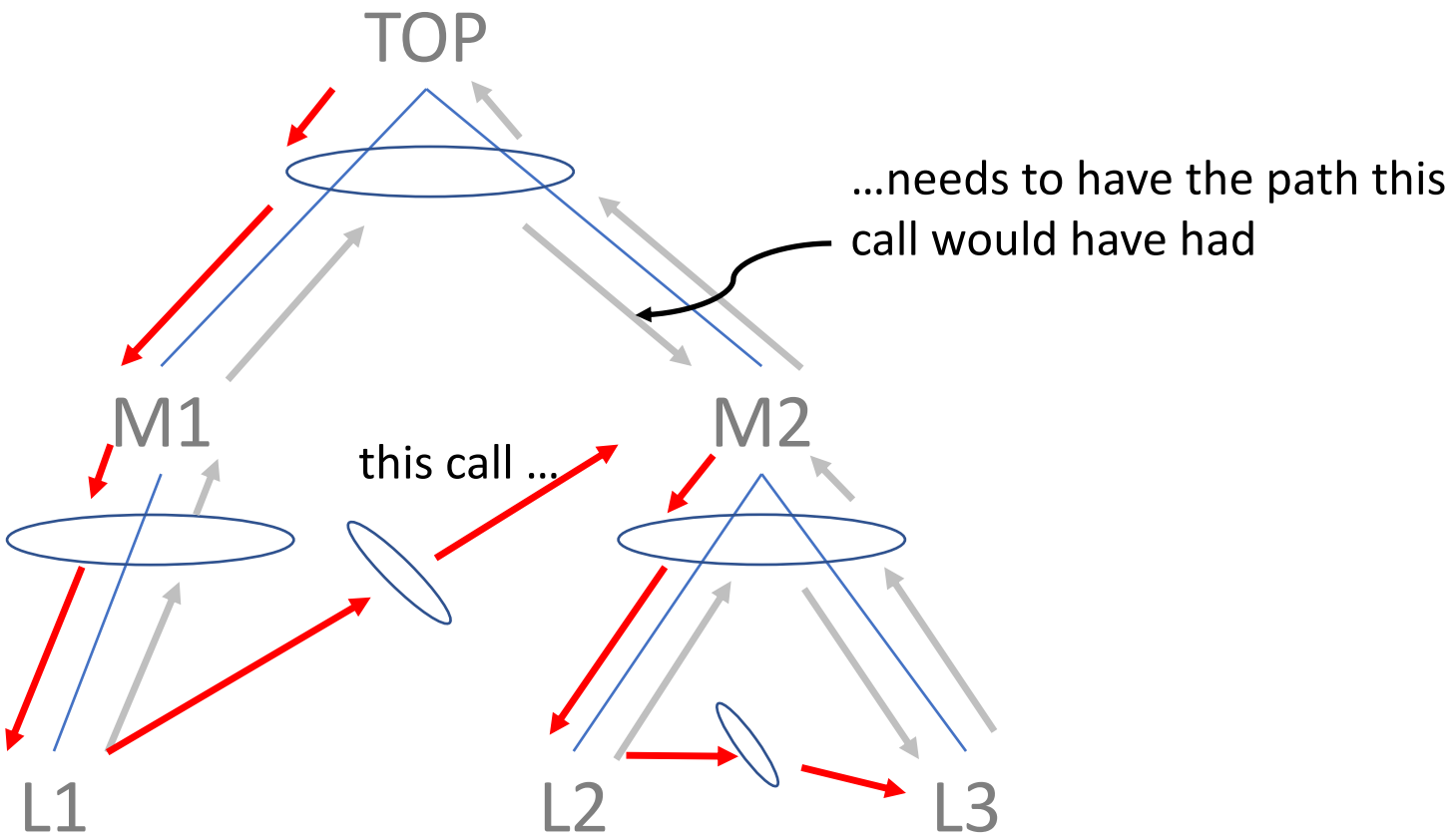


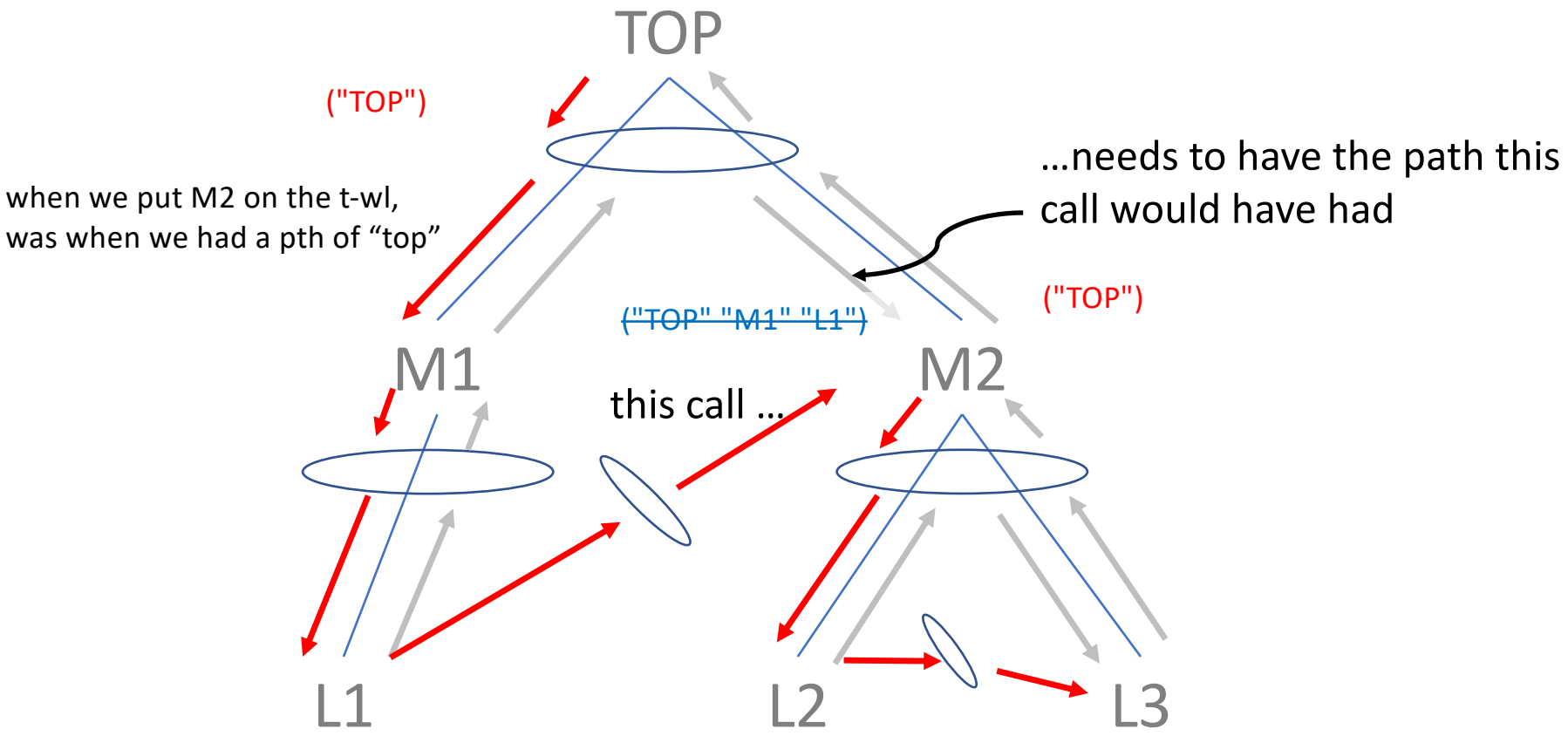
fill in visited at calls to fn-for-t



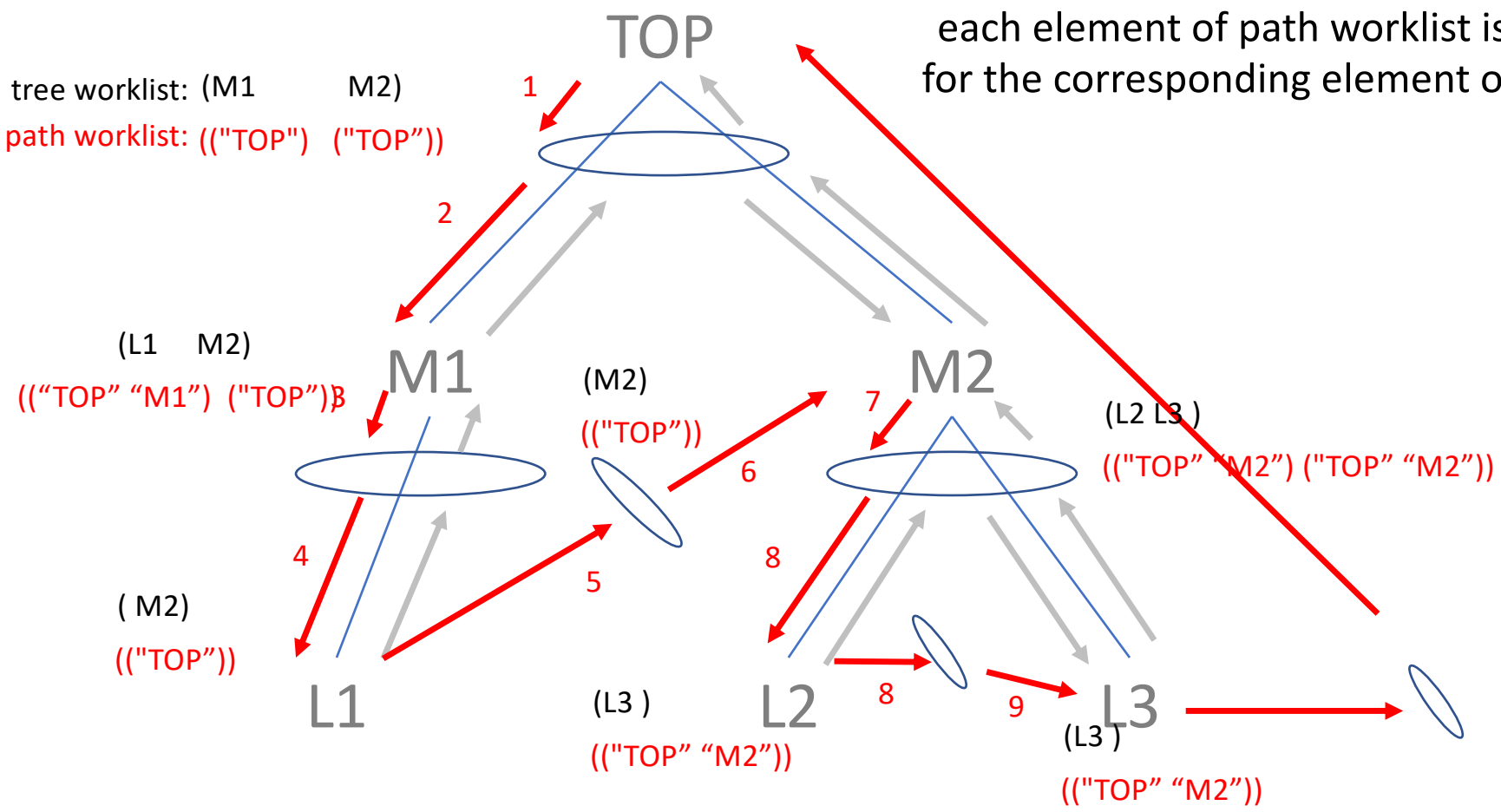
fill in visited at calls to fn-for-t




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,
was when we had a pth of “top”
…needs to have the path this
call would have had
("TOP")
("TOP" "M1" "L1")
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



tail recursion with tandem worklists (tree and path)

```
(@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)))
```