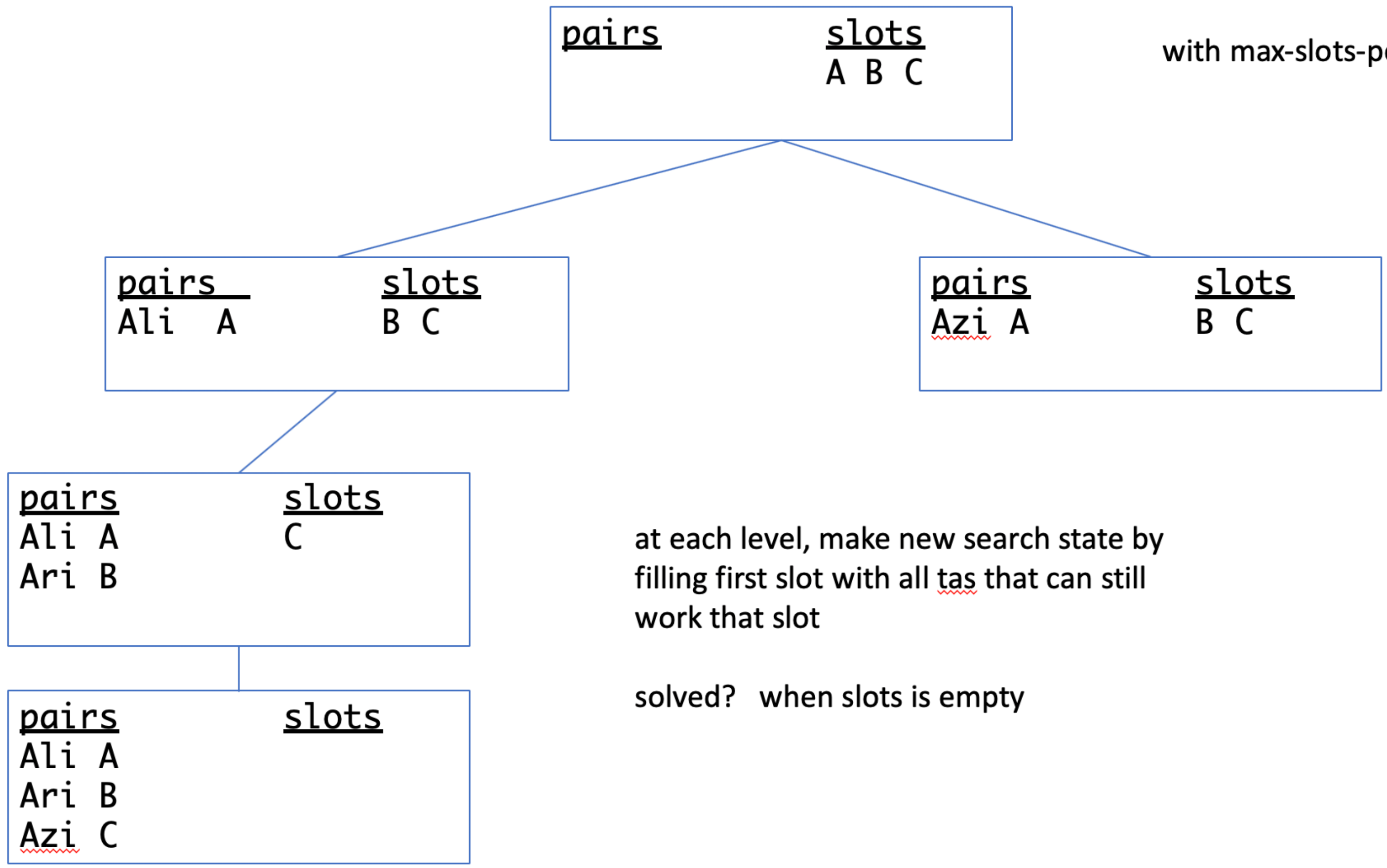
Pset 9

with max-slots-per-ta = 1



at each level, make new search state by filling first slot with all tas that can still work that slot

solved? when slots is empty

```
(define (solve slots tas)
  (local [;;
          ;; trivial case: slots remaining to be scheduled is empty
          ;; reduction step: schedule first slot (or fail to do so)
          ;; argument: slots to schedule is finite, so reducing it by
          ;;           one each time will reach empty

          (define-struct ss (pairs slots))
          ;; SearchState is (make-ss (listof (listof String)) (listof Slot))
          ;; pairs is (list (list "ta name" "lab name")...)  ;a schedule
          ;; slots is the slots that remain to be scheduled

          (define (search/one ss)
            (if (empty? (ss-slots ss))                    ;trivial?
                (ss-pairs ss)                             ;trivial-answer
                (search/list (next-search-states ss))))

          (define (search/list loss)
            (cond [(empty? loss) false]
                  [else
                   (local [(define try (search/one (first loss)))]
                     (if (not (false? try))
                         try
                         (search/list (rest loss))))]))
```

```
;;(@template-origin fn-composition use-abstract-fn)
(define (next-search-states ss)
  (local [(define pairs (ss-pairs ss))
          (define slots (ss-slots ss))
          (define slot  (first slots))]
    (map (lambda (ta)
           ;; assign each available ta to the first slot
           (make-ss (cons (list (ta-name ta) (slot-lab slot)) pairs)
                    (rest slots)))
         (filter (lambda (ta)
                   (and (ta-listed-slot?         ta slot)
                        (ta-has-more-time?       ta      pairs)
                        (ta-not-already-working? ta slot pairs)))
                 tas))))

;; true if TA listed the slot as available time
(define (ta-listed-slot? ta slot)
  (member (slot-lab slot) (ta-avail ta)))

;; true if TA can work more given the current pairs
(define (ta-has-more-time? ta pairs)
  (< (length (filter (lambda (p) (ta-pair? ta p))
                     pairs))
     MAX-SLOTS-PER-TA))

;; true if TA not already assigned to this lab given current pairs
(define (ta-not-already-working? ta slot pairs)
  (empty?
   (filter (lambda (p) (ta-pair? ta p))
           (filter (lambda (p) (slot-pair? slot p))
                   pairs))))
```

Roadmap

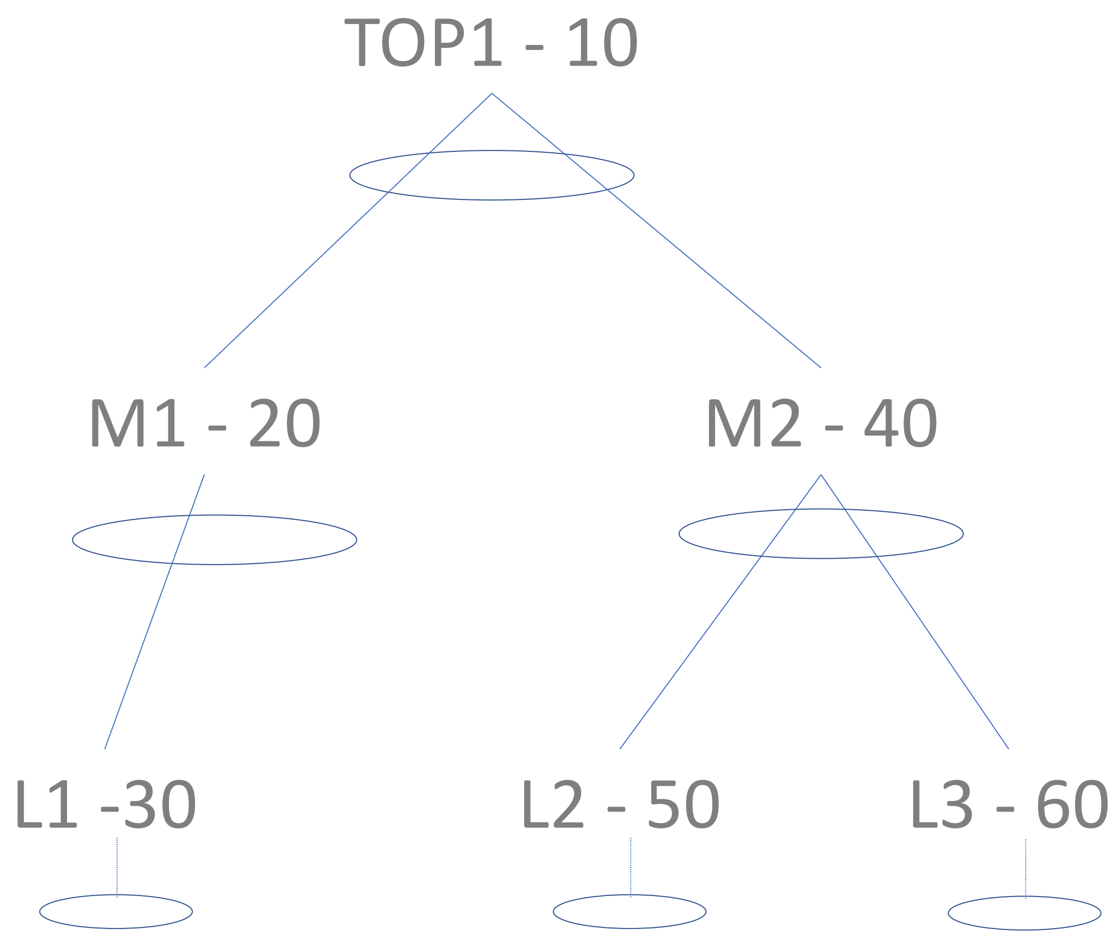
- These five lectures
  - forms of data: trees and graphs
  - recursion: structural non-tail and tail and generative
  - accumulators
    - path in data: previous, upper, lower, pnum, path
    - rsf (result so far)
    - path in tail recursion: vnum, count, leaves, visited
    - worklist
    - tandem worklist

l18 l19 l20 l21 l22

A complicated clicker

- Complex logistics
- Harder problem
- Will ask you to:
  - screen capture picture of tree
  - screen capture clicker question
- Then I will show two functions and you will work on answer
  - graded on participation
  - BUT WORK HARD to try and figure it out
- Then we will work it through together

Screen capture this figure…



Here are the function definitions, without purpose statements, examples, template origins, or accumulator types and invariants.

For each function, work out the progression of accumulator values as the function goes through the entire tree

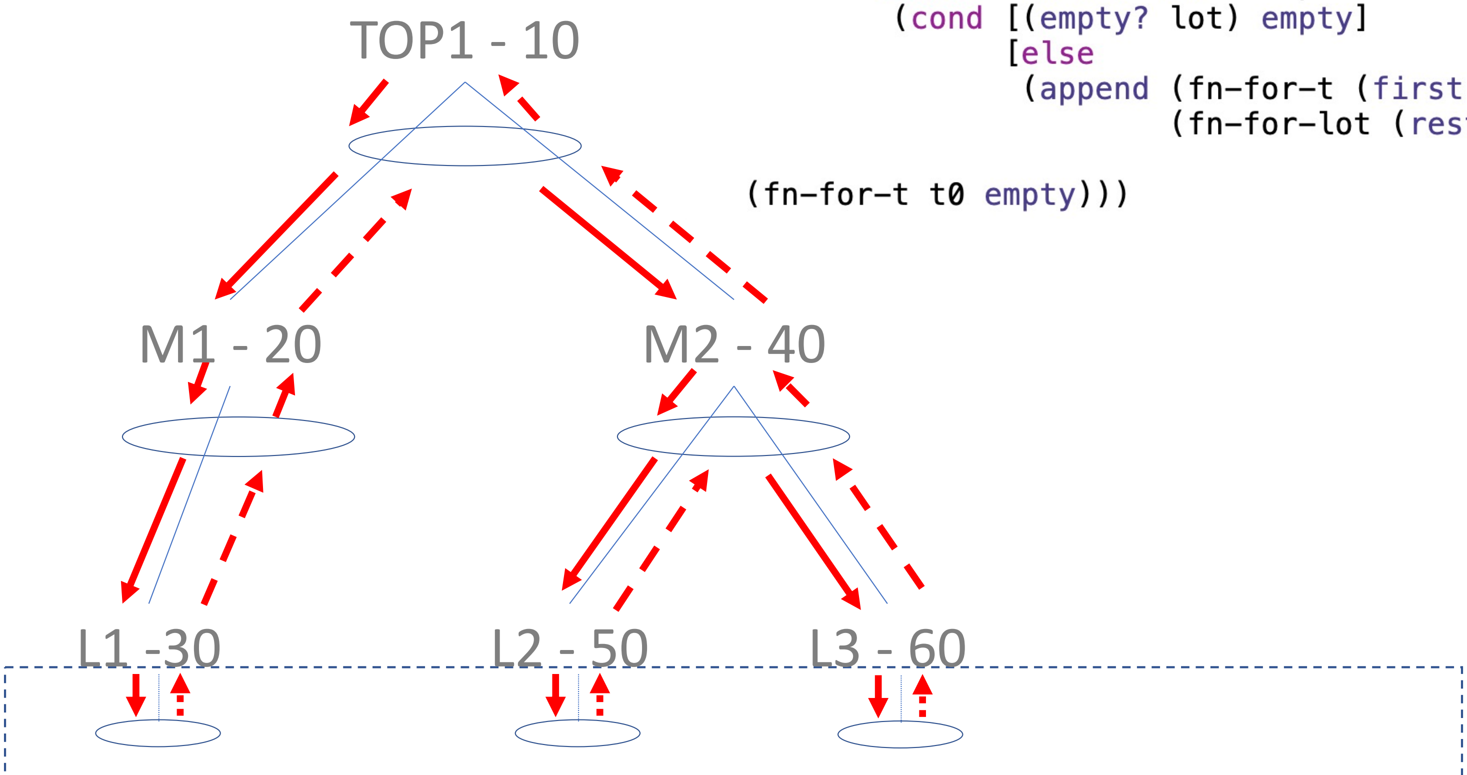
```
(define (foo t0)
  ;; path is ???
  (local [(define (fn-for-t t path)
            (local [(define number (node-number t))  ;unpack the fields
                    (define subs (node-subs t))]      ;for convenience
              (cons (list number (reverse (cons number path)))
                    (fn-for-lot subs (cons number path)))))

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

    (fn-for-t t0 empty)))
```

```
(define (bar t0)
  ;; t-wl is (listof Tree); worklist of Trees to visit
  ;;                        unvisited direct subs of visited trees
  ;; visited is ???
  ;; rsf is ???
  (local [(define (fn-for-t t t-wl visited rsf)
            (local [(define number (node-number t))  ;unpack the fields
                    (define subs (node-subs t))]      ;for convenience
              (fn-for-lot (append subs t-wl)
                          (cons number visited)
                          (cons (list number (reverse (cons number visited)))
                                rsf))))

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

    (fn-for-t t0 empty empty empty)))
```

```
(define (foo t0)
  ;; path is ???
  (local [(define (fn-for-t t path)
            (local [(define number (node-number t))  ;unpack the fields
                    (define subs (node-subs t))]     ;for convenience
              (cons (list number (reverse (cons number path)))
                    (fn-for-lot subs (cons number path)))))

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

    (fn-for-t t0 empty)))
```

foo

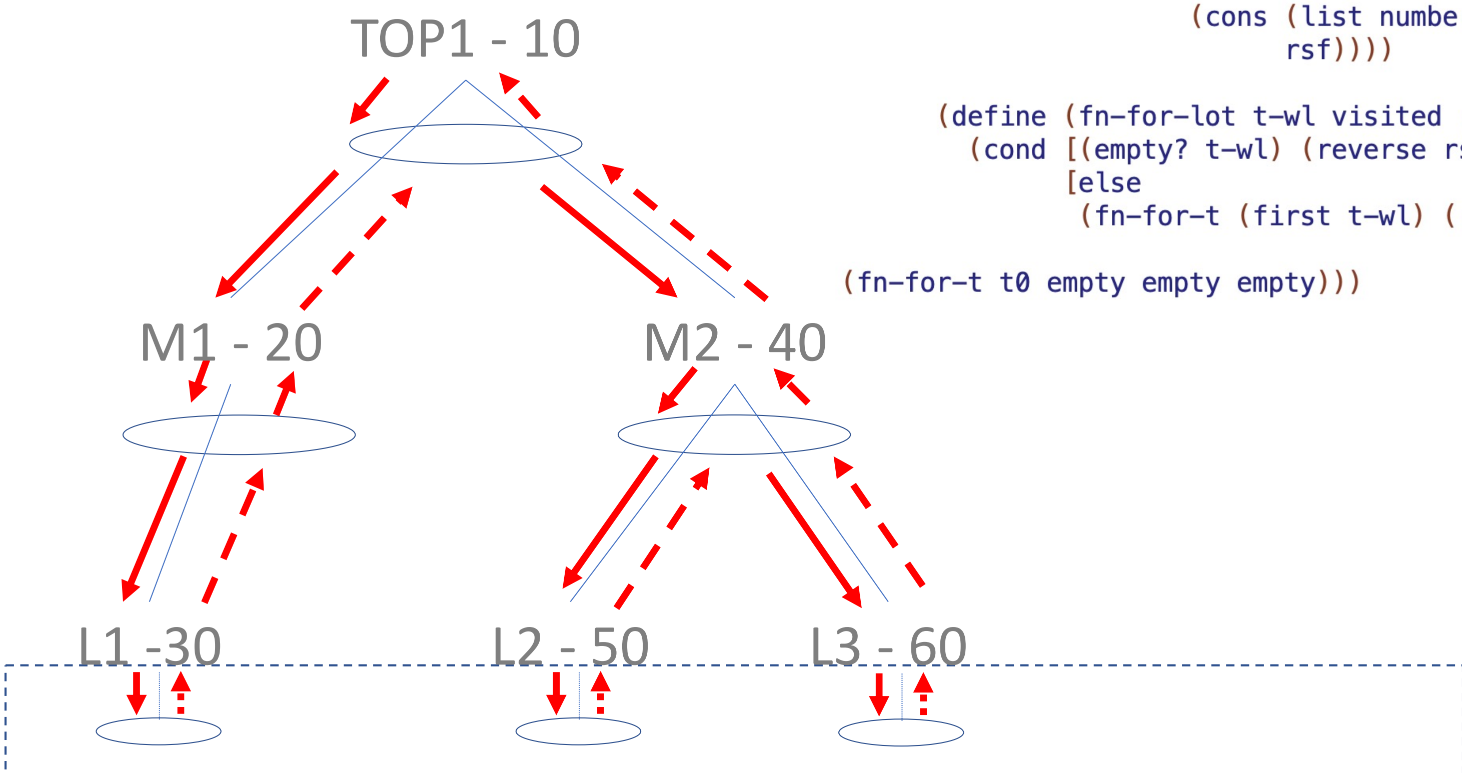
```
(define (foo t0)
  ;; path is ???
  (local [(define (fn-for-t t path)
            (local [(define number (node-number t))  ;unpack the fields
                    (define subs (node-subs t))]     ;for convenience
              (cons (list number (reverse (cons number path)))
                    (fn-for-lot subs (cons number path)))))

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

    (fn-for-t t0 empty)))
```



```
(define (bar t0)
  ;; t-wl is (listof Tree); worklist of Trees to visit
  ;;                        unvisited direct subs of visited trees
  ;; visited is ???
  ;; rsf is ???
  (local [(define (fn-for-t t t-wl visited rsf)
            (local [(define number (node-number t))  ;unpack the fields
                    (define subs (node-subs t))]     ;for convenience
              (fn-for-lot (append subs t-wl)
                          (cons number visited)
                          (cons (list number (reverse (cons number visited)))
                                rsf))))

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

    (fn-for-t t0 empty empty empty)))
```

bar

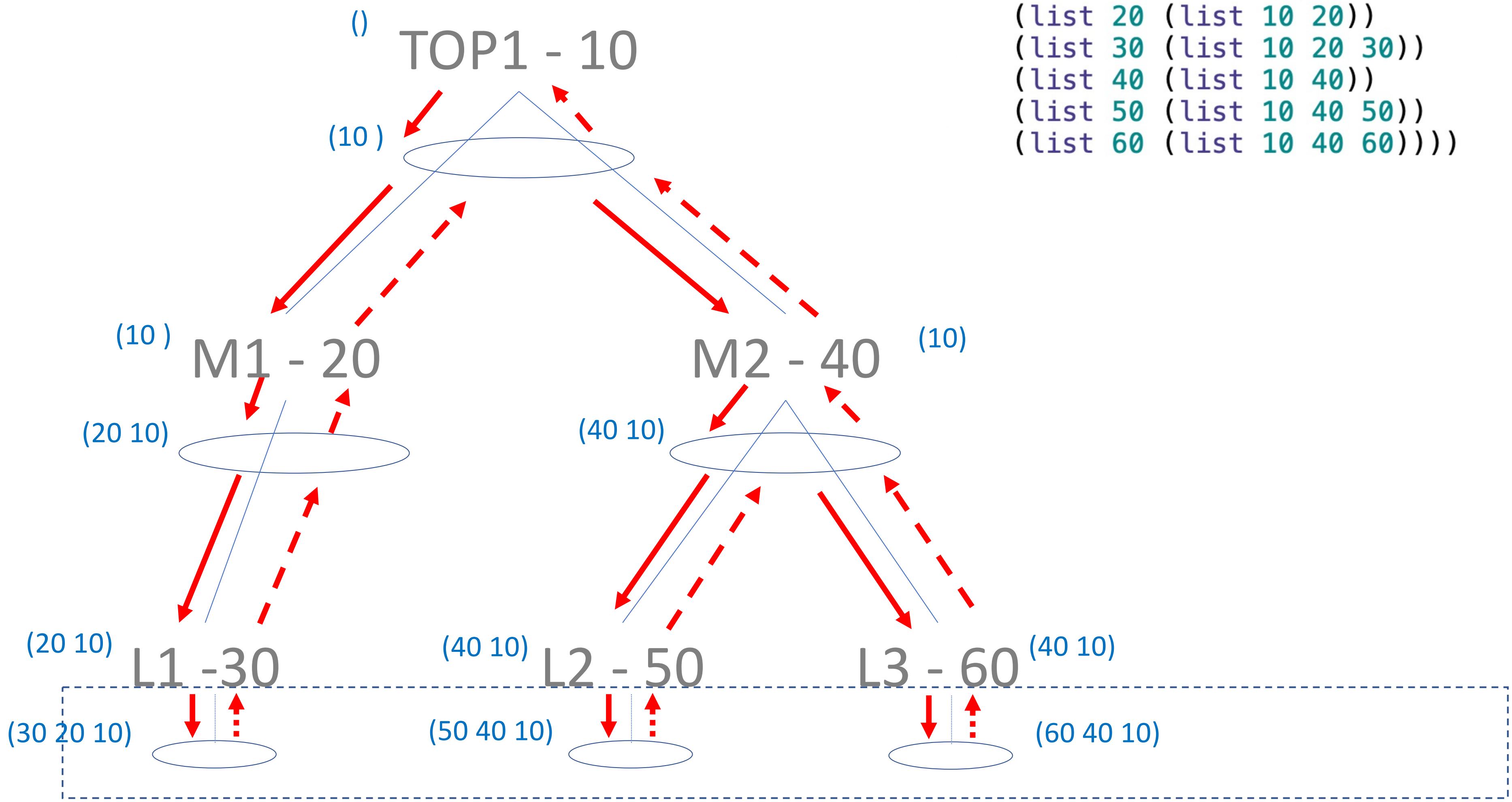
```
(define (bar t0)
  ;; t-wl is (listof Tree); worklist of Trees to visit
  ;;                        unvisited direct subs of visited trees
  ;; visited is ???
  ;; rsf is ???
  (local [(define (fn-for-t t t-wl visited rsf)
            (local [(define number (node-number t))  ;unpack the fields
                    (define subs (node-subs t))]     ;for convenience
              (fn-for-lot (append subs t-wl)
                          (cons number visited)
                          (cons (list number (reverse (cons number visited)))
                                rsf))))

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

    (fn-for-t t0 empty empty empty)))
```



```
;;
;; Which function passes this test?
;;

(check-expect (??? TOP1)
              (list (list 10 (list 10))
                    (list 20 (list 10 20))
                    (list 30 (list 10 20 30))
                    (list 40 (list 10 40))
                    (list 50 (list 10 40 50))
                    (list 60 (list 10 40 60))))

;; A) foo
;; B) bar
;; C) neither
```

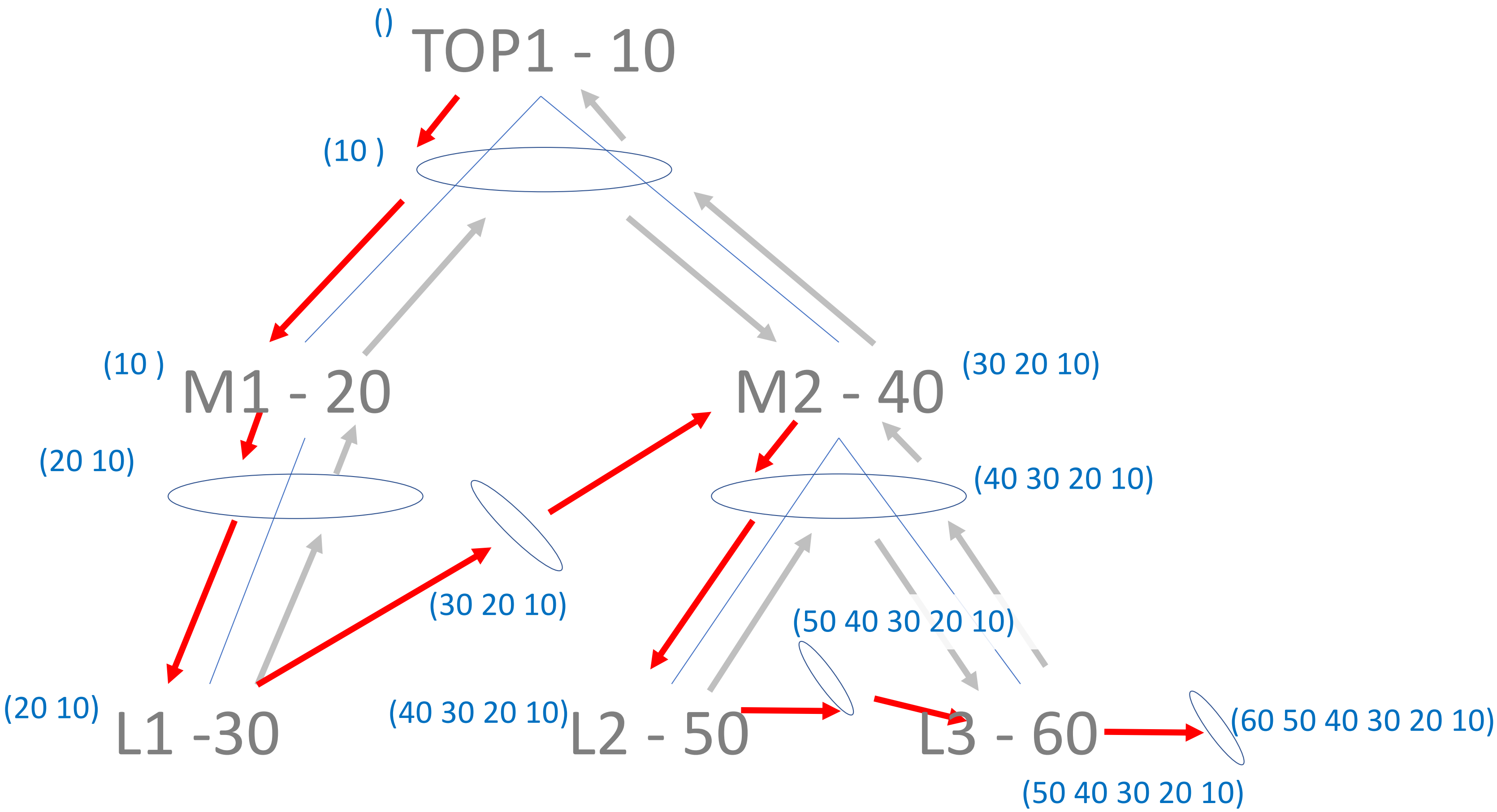
```
;;
;; Which function passes this test?
;;

(check-expect (??? TOP1)
              (list (list 10 (list 10))
                    (list 20 (list 10 20))
                    (list 30 (list 10 20 30))
                    (list 40 (list 10 20 30 40))
                    (list 50 (list 10 20 30 40 50))
                    (list 60 (list 10 20 30 40 50 60))))


;; A) foo
;; B) bar
;; C) neither
```

~~foo~~ produce-path-at-nodes




```
(list (list 10 (list 10))
      (list 20 (list 10 20))
      (list 30 (list 10 20 30))
      (list 40 (list 10 40))
      (list 50 (list 10 40 50))
      (list 60 (list 10 40 60)))
```

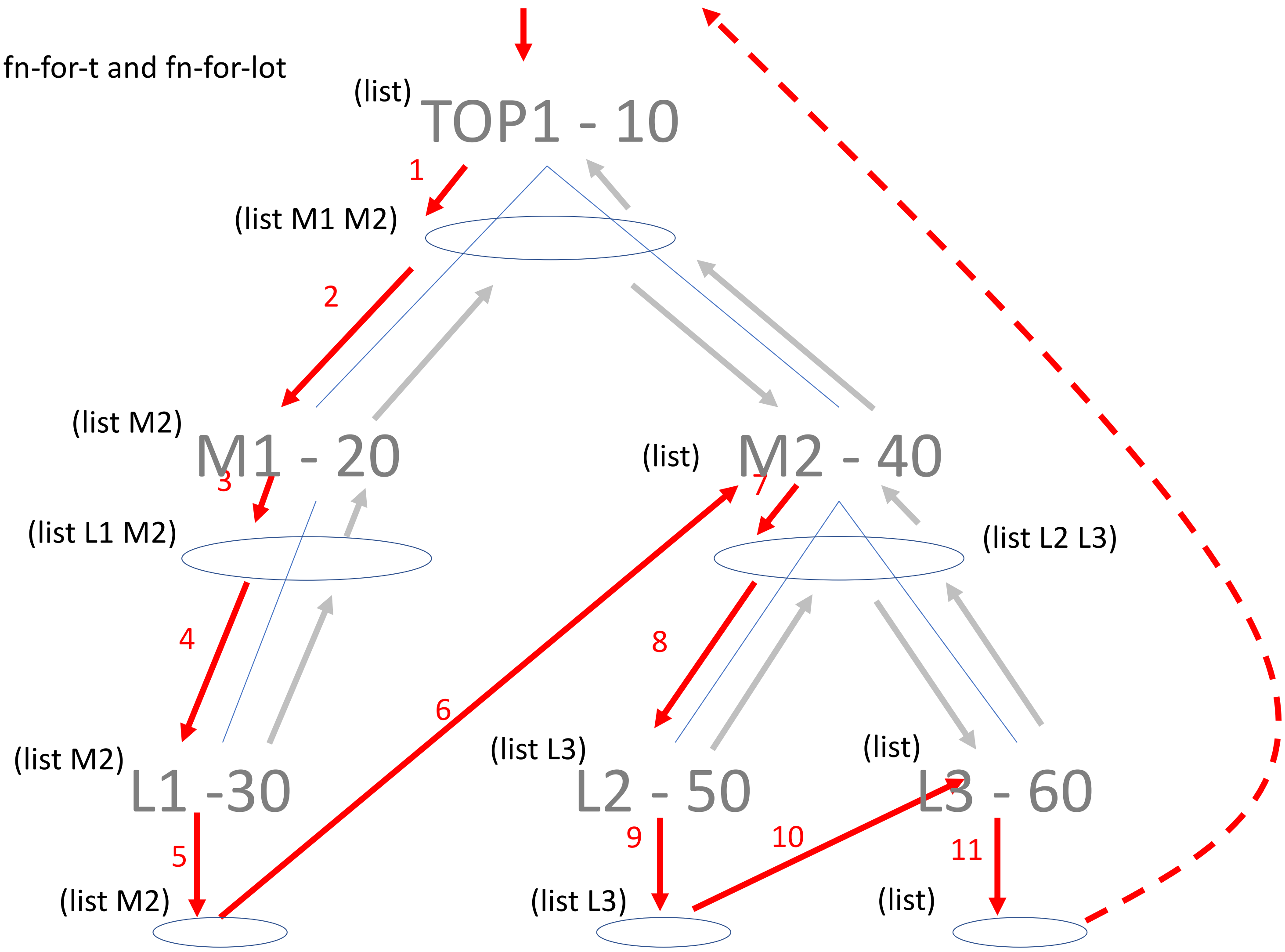
~~bar~~ produce-visited-at-nodes



What’s in the past depends on the recursion

- tail recursion means current call can have all preceding calls
- 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

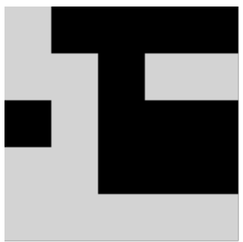
t-wl at fn-for-t and fn-for-lot

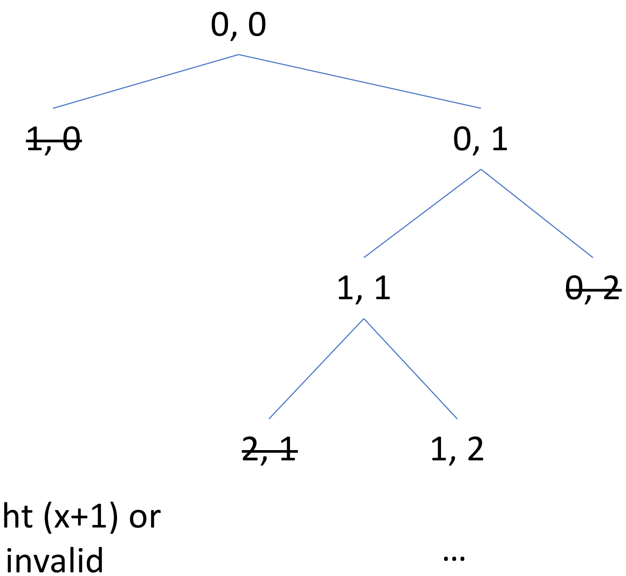


Module 11 - Graphs

- 3 examples
  - lecture – 4 way maze
  - lab – city map
  - problem set – secret castle

Tree of x,y positions moving through this maze





At each step it is only possible to move right (x+1) or down (y+1). But sometimes those may be invalid because they run into a wall or off the edge of the maze.

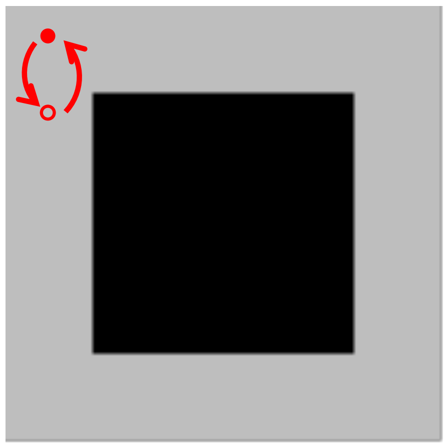
Do not assume each position can have only one valid next position. In general it is an arbitrary-arity tree.

This maze is solveable, so will eventually reach 4, 4. Yay!

```
(define M4
  (list O O O O O
        O W W W O
        O W O O O
        O W O W W
        W W O O O))
```

must move left

need to be able to move up down left right



```
(define M7
  (list O O O O O O O O O O
        W W O W W O W W W O
        O O O W W O W O O O
        O W O O W O W O W W
        O W W O W W W O O O
        O W W O O O W W W O
        O W W O W O W O O O
        O W W O O O W O W W
        O O O O W W W O O O
        W W W W W O O W W O))
```

```
(define M7
  (list O O O O O O O O O O
        W W O W W O W W W O
        O O O W W O W O O O
        O W O O W O W O W W
        O W W O W W W O O O
        O W W O O O W W W O
        O W W O W O W O O O
        O W W O O O W O W W
        O O O O W W W O O O
        W W W W W O O W W O))
```

cycle

How do we prevent going in circles forever?

```
;; structural recursion, with path accumulator

;; trivial:   reaches lower right, previously seen position
;; reduction: move up, down, left, right if possible
;; argument:  maze is finite, so moving will eventually
;;            reach trivial case or run out of moves

;; path is (listof Pos); positions on this path through data
(define (solve/p p path)
  (cond [(solved? p) true]
        [(member p path) false]
        [else
         (solve/lop (next-ps p)
                    (cons p path))]))

(define (solve/lop lop path)
  (cond [(empty? lop) false]
        [else
         (local [(define try (solve/p (first lop) path))]
           (if (not (false? try))
               try
               (solve/lop (rest lop) path)))]))
```

```
(define M7
  (list O O O O O O O O O O
        W W O W W O W W W O
        O O O W W O W O O O
        O W O O W O W O W W
        O W W O W W W O O O
        O W W O O O W W W O
        O W W O W O W O O O
        O W W O O O W O W W
        O O O O W W W O O O
        W W W W W O O W W O))
```

cycle

Would it also work with tail recursion and visited?