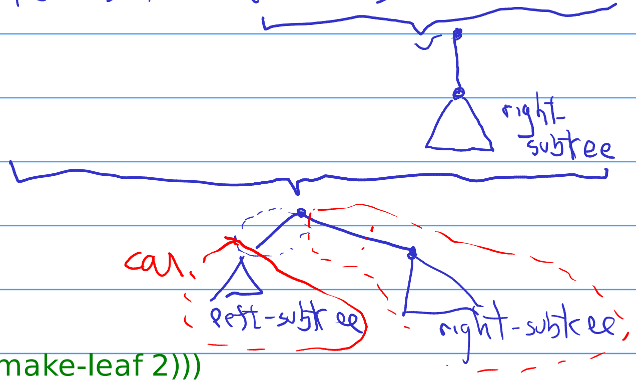
(cons 'b ⸻ a)

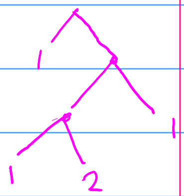
(a) ≡ ⸻ a

```scheme
; some data

(define (make-bin-tree left-subtree right-subtree)
  (list left-subtree right-subtree))
```

— (cons left-subtree (cons rightsubtree '()))




```scheme
(define (left-subtree bin-tree)
  (car bin-tree))

(define (right-subtree bin-tree)
  (cadr bin-tree))

(define (make-leaf a) a)


(define tree-010 (make-bin-tree (make-leaf 1) (make-leaf 2)))
(define tree-01 (make-bin-tree tree-010 (make-leaf 1)))
(define tree-0 (make-bin-tree (make-leaf 1) tree-01))

(define tree-1001 (make-bin-tree (make-leaf 2) (make-leaf 1)))
(define tree-100 (make-bin-tree (make-leaf 2) tree-1001))
(define tree-101 (make-bin-tree (make-leaf 1) (make-leaf 2)))
(define tree-10 (make-bin-tree tree-100 tree-101))

(define tree-111 (make-bin-tree (make-leaf 1) (make-leaf 2)))
(define tree-11 (make-bin-tree (make-leaf 1) tree-111))
(define tree-1 (make-bin-tree tree-10 tree-11))

(define tree (make-bin-tree tree-0 tree-1))


; suggested exercise: draw this tree and explain the notation
```



; replace-nth

; Here is a tree recursion somewhat more complicated than those we have looked at until now

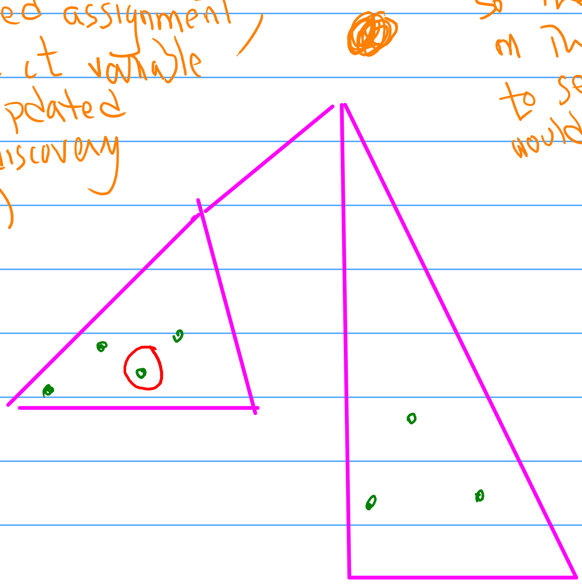
; develop and certify a scheme program replace-nth which takes as input

; a list lst, not necessarily a list of atoms ;;; why rule out atomic input?
; a positive integer, n
; an atom, old
; an atom, new

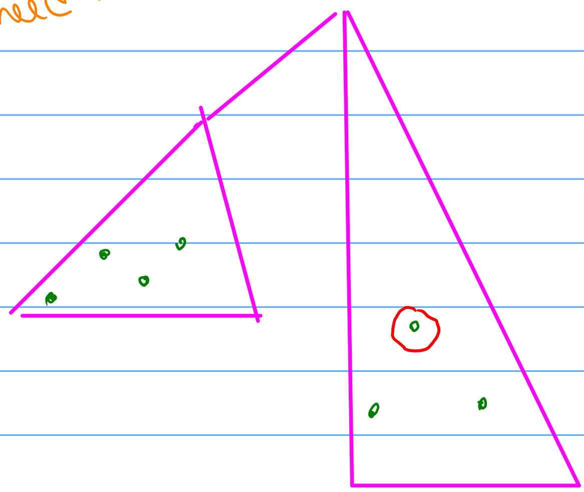
; (replace-nth lst n old new) should replace the nth occurrence of old in
; lst by new (and leave everything else unchanged)

If we allowed assignment a global variable could be updated with each discovery of old (•)

So the recursive call on the cdr — being able to see this global would not need its n parameter to be adjusted.



case (i) – the nth occurrence lies in the car subtree



case (ii) – the nth occurrence lies in the cdr subtree

This variable would permit communication between the recursive calls (on car and cdr) — something functional programming does not allow!

The orange text suggests that an assignment based solution would be significantly more efficient than the one I give on the next page.

```
; here is one idea -- can you explain what is going on?  can you show
; that the code is correct?  can you think of an alternative design?


(define atom?
  (lambda (x)
    (and (not (null? x)) (not (pair? x)))))

(define count
  (lambda (tree a)
    (cond ((null? tree) 0)
          ((atom? tree) ".....")
          ((atom? (car tree))
            (cond ((eq? (car tree) a)
                   (+ 1 (count (cdr tree) a)))
                  (else (count (cdr tree) a))))
          (else
            (+ (count (car tree) a)
               (count (cdr tree) a))))))

(define replace-nth
  (lambda (tree old n new)
    (cond ((null? tree) tree)
          ((atom? tree) ".....")
          ((atom? (car tree))
            (cond ((eq? (car tree) old)
                   (cond ((= n 1) (cons new (cdr tree)))
                         (else (cons old (replace-nth (cdr tree) old (- n 1) new)))))
                  (else (cons (car tree) (replace-nth (cdr tree) old n new)))))
          (else
           (cond ((<= n (count (car tree) old))
                  (cons (replace-nth (car tree) old n new)
                        (cdr tree)))
                 (else
                  (cons (car tree)
                        (replace-nth (cdr tree) old (- n (count (car tree) old)) new)))))))))


     (replace-nth tree 1 5 3)
```

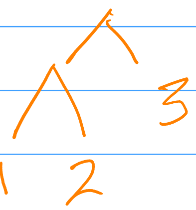
Proofs of both count and replace-nth are by structural induction, using the car-cdr structure of trees

; we can use let to avoid the second call (count (car tree) old) as follows:

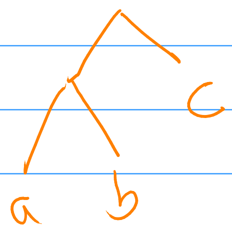
```
(define replace-nth
  (lambda (tree old n new)
    (cond ((null? tree) tree)
          ((atom? (car tree))
           (cond ((eq? (car tree) old)
                  (cond ((= n 1) (cons new (cdr tree)))
                        (else (cons old (replace-nth (cdr tree) old (- n 1) new)))))
                 (else (cons (car tree) (replace-nth (cdr tree) old n new)))))
          (else
           (let ( (m (count (car tree) old))  )
             (cond ((<= n m)
                    (cons (replace-nth (car tree) old n new)
                          (cdr tree)))
                   (else
                    (cons (car tree)
                          (replace-nth (cdr tree) old (- n m) new)))))
           ))))
```

Happily, it is NOT the case that this is the best functional programming can do on this problem.

Suggested: solve this problem by writing a function new-fringe which inputs a tree and a list (the list contains only atoms — it is to be the new fringe)

For example, (new-fringe 

'(a b c)) =

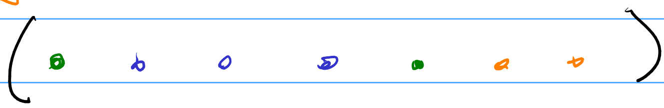


— geometrically unchanged but with fringe '(a b c) rather than '(1 2 3).

(Hint: use the functions given in the first set of list exercises. As well as the fringe function.)

Once we have such a function, one obtains the new fringe from the old fringe by simple flat-list operations.

eg if $n=2$ and the old fringe is



Then the new fringe is