Class 19 April 4 2024

① append & reverse

; append — a primitive

```
(define (myappend l m)
  (cond ((null? l) m)
        (else (cons (car l) (myappend (cdr l) m)))))
```

l, m are both lists *

example: (myappend '(1 2) '(3 4 5)) = (1 2 3 4 5)

algebraically: '() is the zero — ie —

(myappend '() '(3 4 5)) =

(myappend '(3 4 5) '()) = (3 4 5)

not set union — look in A&S ch 2 for discussion of representing sets as lists without duplicates — and then observe (append '(1) '(1 2)) = (1 1 2).

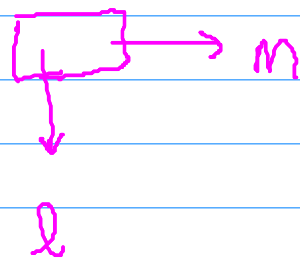
observe: cdrs down first list, does nothing with the second list.

* weirdness: in fact, the second argument can be any scheme object — not required even to be a pair.

Runtime: linear in the length of the first argument

Can the efficiency be improved? — eg — by selecting the shorter list for cdring down. Possibly — depending on the data set, might it be worth incurring the cost of computing (length l) and (length m)?

Let's think about how this might work – noting that the result we want is not



we might try to build a solution using snoc :

"cons", backwards

element

list

```
(define (snoc e l)
  (cond ((null? l) (list e))
        (else (cons (car l)
                    (snoc e (cdr l))))))
```

Clearly, we can compute (append l m) by "snocing each element of m down l". But could this ever be more efficient than the design we gave first, regardless of the relative lengths of l and m? I'll leave this for you to think about.

We sketch an induction proof for append ...

What about an iterative append?

```
(define (append-iter l m)
   (define (iter remaining so-far)
      (cond ((null? remaining) so-far)
            (else (iter (cdr remaining)
                        (cons (car remaining) so-far)))))
   (iter (reverse l) m))
```

GI: (append L M) =
(append (reverse remaining) so-far)

math functions

Finally, for append, we note

algebraically — append is NOT commutative —
(append l m) ≠ (append m l)

algebraically — append IS associative

; reverse — another primitive in R^5RS

Recursive:

```
(define (myreverse l)
  (cond ((null? l) l)
        (else (append (myreverse (cdr l)) (list (car l))))))
```

work out the pf!
IH: rec. call returns the reverse of (cdr e)

observe

math functions

reverse (L) =

append (reverse (cdr L)),
list (car L))

Feel free to use the math functions of the same names as your scheme functions in your writeups!

; quadratic time, and we wonder whether we can do better

does everyone see this? If $|\ell| = n$, then the first call to append needs time $(n-1)$ the second needs time $(n-2)$, and so on.
You recall $\sum_{i=1}^{k} i = \Theta(k^2)$

```scheme
(define (myreverse l)
  (define (iter l result)
    (cond ((null? l) result)
          (else (iter (cdr l) (cons (car l) result)))))
  (iter l '()))

; linear time, so better
```

again: math functions

GI: reverse(L) = append((reverse (cdr l)), result)

- This GI is of the form

  total work = work remaining + work done.

(2)

; map — a primitive, but note that the R5RS map primitive is more powerful than my mymap - function, compatible with elts of the list l

```
(define (mymap f l)
  (cond ((null? l) l)
          (else (cons (f (car l)) (mymap f (cdr l))))))

(define (square x) (* x x))

; examples ...
```

look up map-reduce

(mymap square '(1 2 3 4)) =

((square 1) (square 2) (square 3) (square 4)) =

(1 4 9 16)

observe: (a) |(mymap f l)| = |l| map does not change length

(b) order is preserved - ie - the i^{th} element of the returned list is $(f\ l_i)$, where l_i is i^{th} elt of the input list.

(c) linear time, if f works in constant time - more generally, if $(f\ l_i)$ is constant for each l_i, then the time needed is $time_{(f\ l_i)} * |l|$.

3

```
; accumulate

(define (accumulate op init seq)
  (cond ((null? seq) init)
        (else (op (car seq) (accumulate op init (cdr seq))))))

; accumulate is also known by the name foldr
```

binary function

list

(fold-right)

```
; examples ...
```

(accumulate + 0 '(1 2 3 4)) =

(+ 1 (accumulate + 0 '(2 3 4)))

returns (+ 2 (+ 3 (+ 4 0)))

returns

(+ 1 (+ 2 (+ 3 (+ 4 0))))

(accumulate * 1 '(1 2 3 4)) =

$$\prod_{i=1}^{4} i$$

note that the initial values are (and need to be) different.

(accumulate cons '() '(1 2 3 4)) =

(1 2 3 4)

(accumulate append '() seq) ... →

what type is seq?

got to be a list of elements compatible with append — ie — a list of lists.

(accumulate append '() '((1 2 3) (4 5) (6))

=

(1 2 3 4 5 6)

We say that the input list has been <u>flattened</u>.

Try: (accumulate append '(a b c) '((1 2 3) (4 5)))

we've changed init

Try: (accumulate append '()
'(((1 2 3)) ((4 5)) (6))

(define (flatten list-of-lists)
  (accumulate append '() list_of_lists))

you'll see that flatten "promotes" each sublist by 1 level (thinking of trees)

Note That accumulate does not necessarily preserve The length (etc) of The input list. ITS not even required to return a list.

```
; examples ...
```

$$\boxed{\begin{array}{l}\texttt{(define (mymap f l)}\\ \texttt{ (accumulate (lambda (x y) (cons (f x) y)) '() l))}\end{array}}$$

op: (lambda (x y)
  (cons (f x) y))

init: '()

seq: l

```
;; illuminating

(define (leftmost lst)
  (foldr (lambda (x y) x) '() lst))
```

ie, accumulate

can you apply The same reasoning to explain why This is called leftmost?

To understand why this implements map, you need only realize that ~~in~~ all of the examples above, The second argument to op is The result of The recursive call — of accumulate on (cdr seq)

Try computing

$\texttt{(accumulate}\ \underbrace{\texttt{(lambda (x y) (+ x y))}}_{\text{This is just The binary specialization of +}}\ \texttt{0 '(1 2 3 4))}$

(same as

(accumulate + 0 '(1 2 3 4)))

Sibling function to foldr :

```
; accumulate-left, or foldl

(define (accumulate-left op init seq)
  (define (iter acc rest)
    (if (null? rest)
        acc
        (iter (op acc (car rest))
              (cdr rest))))
  (iter init seq))
```

eg, the accumulated sum

the first of the remaining addends in (accumulate-left + 0 '(1 2 3 4))

at a typical intermediate point, acc might be 1+2

$\boxed{1 \quad 2}$ 3 4

and next we add 3.

init is used first and not last

```
; examples ...

(define (elem e l)
  (accumulate-left
   (lambda (acc first) (or acc (eq? first e)))
   #f
   l))

(elem 3 '(1 2 3 4 5))
(elem 6 '(1 2 3 4 5))

; check out the symmetry - compare leftmost, above

(define (rightmost lst)
  (accumulate-left (lambda (x y) y) '() lst))
```

ie, e ∈? l

when eq? is the right test

use equal? for a more general version.

This is op

acc initially #f, so first: (or #f (eq? 3 1))

So on the second call to iter, acc = (or #f #f) = #f

and on the third call, acc = (or #f (eq? 3 2)) = #f

and on the forth call, acc = (or #f (eq? 3 3)) = #t

and once acc is #t, it's value does not change (even though we don't have early exit)

---

why bother with the folds? Their importance and usefulness are hard to overstate: virtually all functions which process lists element-by-element can be realized as folds.

④ filter

not a primitive

a unary boolean function — compatibility is required

a list

```
; filter

(define (filter pred seq)
  (cond ((null? seq) seq)
        ((pred (car seq)) (cons (car seq) (filter pred (cdr seq))))
        (else (filter pred (cdr seq)))))

; examples ...
```

(filter odd? '(1 2 3 4)) = (1 3)

iterative or recursive?

Recursive, because the stack might be used.

---

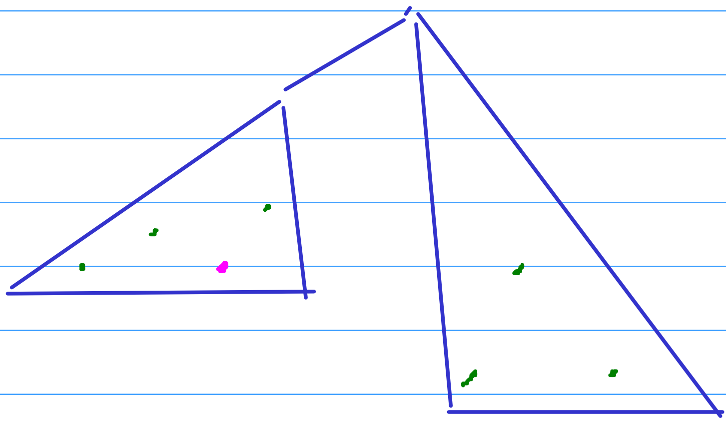
suggested for memorization

- accumulate
- filter

You need these at your fingertips if they are to be useful in conceptualizing problem solutions.

5

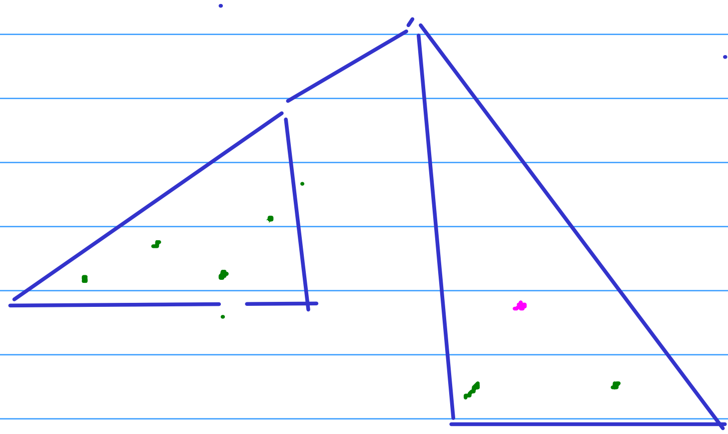
Brief comment on the replace-n^{th}-occurrence problem (of old by new)



Two cases - either occurs in the car or occurs in the cdr.

cdr is the interesting case, since there is no way for the recursive call on the car to communicate to the recursive call on the cdr the number of olds which were found in the car

t



The 6^{th} occurrence of old in t is the 2^{nd} occurrence of old in (cdr t)

If we had assignment to a mutually visible global variable, this would be trivial!

eg) Hint: count the number of occurrences of old in the car (in a separate tree traversal) - and adjust the replacement index for the cdr call accordingly.