Higher Order fns

- functions that manipulate functions

Tail Calls & Tail Recursion

- no loops in fn lang
- normall call nests stack
- tail call pops local frame before calling, like a non-local goto to another fn
- tail calls translated into loops by optimizer

NON TAIL CALL

```
(def (fac n)
   (if (< n 1) 1
      (* n (fac (- n 1)))
))
```

TAIL CALL uses worker inner fn "accumulator style"

```
(def (fac n)
   (def (facc n a)
      (if (< n 1) a
         (facc (- n 1) (* n a)))
   ))
   (facc n 1)
```

evaluate

```
(fac 4) = (* 4 (fac 3))
        = (* 4 (* 3 (fac 2)))
        = (* 4 (* 3 (* 2 (fac 1))))
        = (* 4 (* 3 (* 2 (* 1 (fac 0)))))
        = (* 4 (* 3 (* 2 (* 1 1))))
        = (* 4 (* 3 (* 2 1)))
        = (* 4 (* 3 2))
        = (* 4 6)
        = 24
```

deep stack

```
(fac 4)
   = (facc 4 1)
   = (facc 3 4)
   = (facc 2 12)
   = (facc 1 24)
   = (facc 0 24)
   = 24
```

Fibonacci

HOF (2)

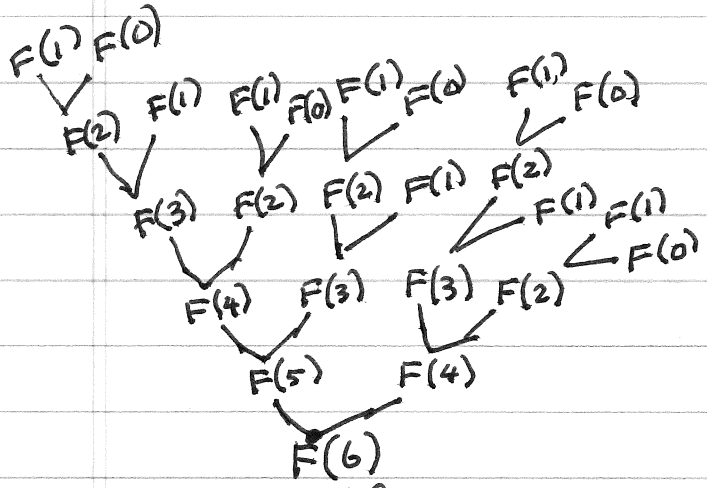
non tail call

```
(def (Fib n)
    (if (< n 2) n
        (+ (Fib (- n 1))
           (fib (- n 2)))
))
```

F(6) = makes 24 calls

F(7) makes 38 calls

$O(2^n)$ calls



call tracetree

tail call

```
(def (fib n)
    (def (fibb n a b)
        (if (< n 1) a
            (fibb (- n 1)
                  b (+ a b))))
    (fibb n 0 1)
)
```

(fib 5) = (fibb 5 0 1)
= (fibb 4 1 1)
= (fibb 3 1 2)
= (fibb 2 2 3)
= (fibb 1 3 5)
= (fibb 0 5 8)
= 5

O(n) calls

Functions on Lists

non-tail calls

```
(def (len l)
     (if (null? l) 0
         (+ 1 (len (cdr l)))))
(def (sum l)
     (if (null? l) 0
         (+ (car l) (sum (cdr l)))))
```

stack usage O(n) to length of list

TAIL CALLS (accumulator style)

```
(def (len l)
     (def (lenn ll n)
          (if (null? ll) n
              (lenn (cdr ll) (+ 1 n))))
     (lenn l 0))
(def (sum l)
     (def (summ ll s)
          (if (null? ll) s
              (summ (cdr ll) (+ s (car ll)))))
     (summ l 0))
```

stack usage O(1)

prev two examples as boring as

```
for (i=0; i<n; ++i)
  for (p=h; p≠null; p=p→link)
```

Want iterator as in

```
for (i : c)
```

so use Higher order fn

(foldl f u l) = (f(f(f u l_0) l_1) l_2 ....)

left associative

$u + l_0 + l_1 + l_2 + \cdots$

left associative tailcalls

```
(define (foldl f u l)
        (if (null? l) u
                (foldl f (f u (car l)) (cdr l)))
))
```

unit or identity

```
(define (sum l)(foldl + ∅ l))
(define (prod l)(foldl * 1 l))
(define (len l)(foldl (λ(n _)(+ n 1)) ∅ l))
```

(sum '(1 2 3))
= (foldl + ∅ '(1 2 3))
= (foldl + 1 '(2 3))
= (foldl + 3 '(3))
= (foldl + 6 '())
= 6

(len '(1 2 3))
= (foldl λ ∅ '(1 2 3))
= (foldl λ 1 '(2 3))
= (foldl λ 2 '(3))
= (foldl λ 3 '())
= 3

λ expr ignores 2nd arg

possible problem with foldl

(foldl - 0 '(1 2 3)) =

(-(-(- 0 1) 2) 3)

= (- 0 (foldl + 0 '(1 2 3)))

similar problem with division

So:

(foldr f u l) = (f l_0 (f l_1 (f l_2 ⋯) u))

right associative

so (foldr - 0 '(1 2 3) = (-(-(- 1 2) 3) 0)

alternating sum

```
(define (foldr f u l)
        (if (null? l) u
            (f (car l) (foldr f u (cdr l)))))
)
```

NOT TAIL REC.

```
(foldr + 0 '(1 2 3))
        = (+ 1 (foldr + 0 '(2 3)))
        = (+ 1 (+ 2 (foldr + 0 '(3))))
        = (+ 1 (+ 2 (+ 3 (foldr + 0 '()))))
        = (+ 1 (+ 2 (+ 3 0)))
        etc...
```

lots of stack

What if no identity?

```
(def• (minl l) (fold l (λ(x y)(if (< x y) x y))
                       (car l)(cdr l))))
  -- crashes if (null? l)
  -- (minl '()) is meaningless without identity
       ∴ crashes on empty
  -- for reals could use +Infinity
(def (minl l)
     (if (null? l) #f
         (fold l (λ(x y)(if(< x y) x y))
                 (car l)(cdr l))))
     - use (car l) as identity
     - #f to signal error
then: (let ((m (minl l)))
        (if m (do right thing)
              (handle error)))
```

Review

```
(def (foldl f u l)  ;tail recur
     (if (null? l) u
         (foldl f (f u (car l)) (cdr l))))
(def (foldr f u l)  ;NOT tail rec
     (if (null? l) u
         (f (car l) (foldr f u (cdr l)))))
```

MAP (also see Perl's map)

```
(def (map1 f l)
     (if (null? l) '()
         (cons (f (car l)) (map1 f (cdr l)))))
```

```
(map1 sqrt '(4 16 64))
      =(cons 2 (map1 sqrt '(16 64)))
      =(cons 2 (cons 4 (map1 sqrt '(64))))
      =(cons 2 (cons 4 (cons 8 '())))
      ='(2 4 8)
```

```
(def (mapp1 f l)
     (foldr (λ(a d)(cons (f a) d)) '() l))
```

mapping arg

```
(map1 sqrt '(4 16 64))
   =(foldr μ '() '(4 16 64))
   =
```

call this μ next page

cont'd next page

remember $\mu \equiv (\lambda\ (a\ d)\ (cons\ (f\ a)\ d))$

lexical
∴ f = sqrt

$= (foldr\ \mu\ '()\ '(4\ 16\ 64))$ HoF (8)

$= (\mu\ 4\ (foldr\ \mu\ '()\ '(16\ 64)))$

$= (\mu\ 4\ (\mu\ 16\ (foldr\ \mu\ '()\ '(64))))$

$= (\mu\ 4\ (\mu\ 16\ (\mu\ 64\ (foldr\ \mu\ '()\ '()))))$

$= (\mu\ 4\ (\mu\ 16\ (\mu\ 64\ '())))$

$= (\mu\ 4\ (\mu\ 16\ (cons\ 8\ '())))$

$= (\mu\ 4\ (cons\ 4\ (cons\ 8\ '())))$

$= (cons\ 2\ (cons\ 4\ (cons\ 8\ '())))$

$= (2\ 4\ 8)$

Filter (grep in Perl)

$m = [\forall x \in \ell \mid p(x)]$

```
(def (filter p? l)                 ; NOT TAIL-R
     (if (null? l) '()
         (let ((carl (car l))
               (fcdrl (filter p? (cdr l))))
           (if (p? carl)
               (cons carl fcdrl)
               fcdrl))))

(filter (λ(x)(> x 0)) '(-3 8 -4 2))
   ⇒ (8 2)
```

HoF:

```
(def (filter p? l)
     (define (c a d)(if (p? a)(cons a d)d))
     (foldr c '() l))

(filter (λ(x)(> x 0)) '(-3 8 -4 2))
      = (foldr c '() '(-3 8 -4 2))
      = ... = (8 2
```

(or)

```
(def (filter p? l)
     (foldr (λ(a d)(if(p? a)(cons a d)d))
     '() l))
```

append

```
(def (append l m)
     (if (null? l) m
         (cons (car l) (append (cdr l) m))))
```

- let's look @ map...
- same, except use m instead of '()
  and use (λ(x) x) instead of f

```
(def (append l m)
     (foldr (λ(a d)(cons a d)) m l))
```

- expensive fn... avoid if possible
- churns heap O(len l) time O(len l)

Reverse a list

```
(def (reverse l)
     (if (null? l) '()
         (append (reverse (cdr l))
                 (list (car l)))))
```

- very inefficient
- multiple calls to append

$O(n^2)$??

... Reverse

- consider list as 2 stacks

```
while (old ≠ Φ) {
        t = pop old
        push new, t
}
```

--- use accumulator style
--- ∴ left recursive (TAILREC)

```
(def (reverse l)
     (def (rev l m)
          (if (null? l) m
          (rev (cdr l)
          (cons (car l) m))))
     (rev l '()))
```

```
(def (reverse l)
     (foldl (λ (d a)(cons a d))
            '() l))
(def (reverse l)
     (define (snoc d a)(cons a d))
     (foldl snoc '() l))
```

ex: (reverse '(1 2 3))

= (foldl snoc '() '(1 2 3))
= (foldl snoc '(1) '(2 3))
= (foldl snoc '(2 1) '(3))
= (foldl snoc '(3 2 1) '())
= '(3 2 1)

Compose

want $m = [\forall x \in l, (f(g\,x))]$

(map1 f (map1 g l))

- maps g onto each elt of l, makes result
- maps f onto each elt, gen result
- inefficient, intermediate is complete list then gc'd.
- want deforestation

(def (compose f g)
  (λ (x) (f (g x))))

- returns a λ
- ex: want $\sin^2 x \equiv (\sin x)^2$

(define (sinsq x)
  ((compose square sin) x))

- want squares of sin of each $x \in l$

(map1 (λ(x)(compose square sin)) l)

- does not create intermediate (temp) list

(define sinsq (compose square sin))

zip $(l_0 l_1 \ldots)(m_0 m_1 \ldots)$
$\Rightarrow ((l_0 m_0)(l_1 m_1) \ldots)$

- what if lists diff lengths?
- here ignore & use shorter len

```
(def (zip f l m)
     (if (or (null? l)(null? m)) '()
         (cons (f (car l)(car m))
               (zip f (cdr l)(cdr m)))))
```

- not tail recursive
- vague similarity to foldr

Inner product

$$\sum_i l_i m_i$$

```
(define (ip l m)
        (foldl + 0 (zip * l m)))
```

foldl as foldr and viceversa?

(def foldr .... (foldl) ....)

- can't
- foldl is strict in tail
- foldr is not - needs a stack

- all loop can be replaced by tail recursion
- only tail recursion ≡ loop
- non tail recursion needs a stack

```
(def (foldl f u l)
     (foldr (λ (x g) (λ (a) (g (f a x)))) id l u))
```

where

```
(define (id x) x)
```

- but that would be less efficient.
- math interest only

Inefficient qsort

```
(def (qsort <? l)
     [illegible]
     (if (null? l) '()
         (let* ((a (car l))
                (d (cdr l))
                (l< (filter (λ(x)(<? x a)) d))
                (l>= (filter (λ(x)(not(<? x a))) d))
           (append (qsort <? l<)
                   (list a)
                   (qsort <? l>=))))))
```

- Bad pivot?
- churns the heap.
- Qsort works better on vectors, not lists

HOF (16)

```
#!/afs/cats.ucsc.edu/courses/cmps112-wm/usr/racket/bin/mzscheme -qr
;; $Id: mergesort.scm,v 1.2 2014-10-31 17:35:08-07 - - $

(define (foldl fn unit lis)
        (if (null? lis) unit
            (foldl fn (fn unit (car lis)) (cdr lis))))

(define (foldr fn unit lis)
        (if (null? lis) unit
            (fn (car lis) (foldr fn unit (cdr lis)))))

(define (grep ok? lis)
        (define (test hd tl) (if (ok? hd) (cons hd tl) tl))
        (foldr test '() lis))

(define (mergesort <? lis)
        (define (flipflop tf) (lambda (_) (set! tf (not tf)) tf))
        (define (merge lis1 lis2)
                (cond ((null? lis1) lis2)
                      ((null? lis2) lis1)
                      ((<? (car lis1) (car lis2))
                           (cons (car lis1) (merge (cdr lis1) lis2)))
                      (else (cons (car lis2) (merge lis1 (cdr lis2))))))
        (define (msort lis)
                (if (or (null? lis) (null? (cdr lis))) lis
                    (merge (msort (grep (flipflop #t) lis))
                           (msort (grep (flipflop #f) lis)))))
        (msort lis))

```

Parsing a list

expr → token | '(' list ')'
list → expr list | nothing

```
(def (read)
     (let ((token (read-token)))
          (if (eqv? token "(")
              (read-list)
              token)))

(def (read-list)
     (let ((token (read-token)))
          (if (eqv? token ")")
              '()
              (cons (read)(read-list)))))
```

Thunk

passing args unevaluated

(if a b c) – primitive
(cond —) – macro

mechanism

```
(def (choose test consq altern)
     (if (test) (consq) (altern)))
```

```
(choose (λ()(< a b))
        (λ() (/ a b))
        (λ() ( * a b)))
```

λ gets passed unevaluated

– used in lazy evaluation

```
(def (choose T C A)
     (if (T) (C) (A)))
```

Define-syntax

cond is just a macro using if

Use def-syntax to define thunks

```
(define-syntax delay
    (syntax-rules ()
        ((_ exp) (make-promise (λ () exp)))))
(define force
    (λ (promise) (promise)))
(define make-promise
    (λ (promise)
        (let ((val #f) (set? #f))
            (λ (set?) val
                (let ((x (promise)))
                    (if (not set?)
                        (begin (set! val x)
                               (set! set? #t))))))))
```

---

so (f (delay (+ x y))

then in f:

```
(define (f exp)
    ⋮
    (force exp))
```

memoize