Higher Order Perl

Scheme (1)

- R^5RS : 50 pages
- uniform prog/data
- ancestor: Lisp
- interpretive, gcol
- all objs first class, incl fns.
- lists, vectors, structs.
- call by value (applic order)
- core syntactic forms
- lexically scoped
- supports tail calls

Syntax

(atoms...) list
(a . d) pair
(f x y) f applied to x y
predicates end? equal?
imperatives end! set!
fns prefix/infix/type string-append.
conv a->b

Interactive top ~~loop~~ (REPL)

```
mzscheme
> 1
1
> (+ 2 3)        — prefix notation
5
> '(1 2 3)       — quoted form.
(1 2 3)
```

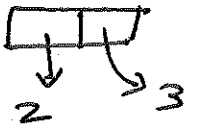
MzScheme can also use [] {}

Scheme ②

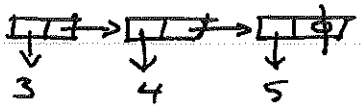
lists

(cons 2 3) ⟹ (2 . 3)

"dotted pair"



(list 3 4 5)



≡ (3 . (4 . (5 . ())))

() is null

(define x '(1 2 3))

(car x) ⟹ 1

(cdr x) ⟹ (2 3)

(cadr x) ⟹ (3)

(caddr x) ⟹ ()

Contents Address Reg

" Decrement "

cdar, ...

(define sq (lambda (x) (* x x)))

(define (sq x) (* x x))

(sq 5) ⟹ 25

only #f is false

Objects

number < exact / inexact

complex

real

rational

integer { bignum / fixnum }

boolean? #t #f

pair?

list?

symbol?

char?

string?

vector?

procedure?

atom?

Scheme (3)

exprs

```
'(2 3)  => 2 3
(2 3)   => error : apply 2 to 3
(+ 2 3) => 5
'(+ 2 3) => (+ 2 3)
(list 'a 'b 'c) => (a b c)
(list a b c) => (20 30 40)
                ~~~ if var has val.
```

Let exprs

```
(let ((x 2))
     (+ x 3)) => 5

(let ((x 2)(y 3))
     (+ x y)) => 5

(let ((f +))
     (f 2 3)) => 5

(let ((+ *))
     (+ 2 3)) => 6   <-- !!
```

+ is a variable.

λ-exprs

```
(lambda (x y) (+ x y))
         ↑args   ↑body
(λ (arg ...) exp ...)

((λ (x) (+ x x)) 4) => 8
```

λ-exp is an object just like any other

Scheme (4)

```
(let ((x 'a))
   (let ((f (λ(y)(list x y))))
        (f 'b)))
```

$\Rightarrow$ (a b)

- free variable
- bound variable

let is just a λ-Expr

```
(let ((x 'a)) (cons x x))
```

$\equiv$ `((λ (x) (cons x x)) 'a)`

- syntactic extension.

`(λ (x y . z) ξ1 ξ2 ...)`

1st, 2nd, rest 3rd ... — 2 or more params.

top level

```
(define (f x) (+ x 1))

(define (cadr x) (car (cdr x)))
≡ (define (a . d) d)
```

Cond

Scheme ⑤

(if test conseq alt)
(if test consequent) → #f ⇒ unspecified.

if ⇒ uses normal order as special form.

(cond clause ...)
clause → (test expr ...)
↑ returns last expr

else.

```
(cond ((> n 0) 'pos)
      ((< n 0) 'neg)
      (else 'zero))
```

```
(define (abs n) (if (< n 0) (- n) n))
```

Recursion

- no loops in lang
- have tail calls.

```
(define (len l)
  (if (null? l)
      0
      (+ 1 (len (cdr l)))))
```

but ⇒ O(n) stack space
∴ bad

accumulator style

Scheme (6)

rewrite tail call → O(1) stack.

```
(define (len l)
  (define (len+ l+ n)
    (if (null? l+)
        n
        (len+ (cdr l+) (+ n 1))))
  (len+ l 0))
```

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

Really bad!!!

O(n) stack
O(n²) time
O(n²) heap

accumulator style

```
(define (rev list)
  (define (rev in out)
    (if (null? in) out
        (rev (cdr in)
             (cons (car in) out))))
  (rev list '()))
```

two stacks

O(n) time
O(n) heap
O(1) stack.

Scheme (7)

ex: Quad.root

$$ax^2 + bx + c = 0$$

$$\Rightarrow \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

```
(define (quadroot a b c)
  (let ((-b (- b))
        (n2a (* 2 a))
        (b24ac (sqrt (- (expt b 2) (* 4 a c)))))
    (cons (/ (+ -b b24ac) n2a)
          (/ (- -b b24ac) n2a))))
```

Lexical vars

```
(define (make-count)
  (let ((n 0))
    (lambda ()
      (let ((v n))
        (set! n (+ n 1))
        v))))
```

set! is bad

it's un-lambda

Scheme (7½)

predicates

eq? - identical

eqv? - objects ~~are~~? equivalent

equal? - structural equality

= - numeric equality

also < <= > >=

~~not to be confused with~~

(< x_1 x_2 x_3 ...) monotonic increas

(>= x_1 x_2 x_3 ...) monotonic decreas

no ≠

char <?

string <?

More recur

```
(define (fac n) (if (< n 1) 0 (* n (fac (- n 1)))))
```

→ O(n) stack.

```
(define (fib n) (if (< n 2) n
                    (+ (fib (- n 1)) (fib (- n 2)))))
```

→ O(2^n) ← BAD

tail rec

accumulator style.

```
(define (fac n)
  (define (facc n r)
    (if (< n 1) r
        (facc (- n 1) (* n r))))
  (facc n 1))
```

M2Scheme BIGNUM

```
(define (fib n)
  (define (fibb n a b)
    (if (< n 2) a
        (fibb (- n 1) b (+ a b))))
  (fibb n 0 1))
```

"priming the pump"

O(n) time
O(1) stack

interface fn
+
worker fn.

let

Scheme 9

(let ((v1 e1)(v2 e2) ...)
  ex1, ex2 ... .)
(let*
(letrec.

let - exprs eval outside of let

let* - exprs in seq.
- each var available in next

letrec - all exprs in scope of all
- values
- allows mutual recursion.

---

define

(define v e)
(define (v v...) e e e e...)
(define (v . v) e e e e...)
(define (v v v... . v) e e e...)

(proc e e e e ...)

Scheme (10)

apply proc to exprs.

(apply + '(4 5)) => 9

(apply min 5 1 3 '(6 8 2 5)) => 1

(apply v ... list)

Sequencing

(begin e_1 e_2 e_3 ...)

exprs eval in sequence L → R.

needed for imperatives

~~HOF~~

Scheme II ①

Tail Calls

```
(def (fac n)                                  O(n) stk
     (if (< n 1) 1                            O(n) time
         (* n (f (- n 1))) ))
(def (fib n)
     (if (< n 2) n                            O(2^n) time
         (+ (fib (- n 1) (- n 2)))))          O(n) stk.
(def (len l)
     (if (null? l) ∅
         (+ 1 (len (cdr l))) ))
```

⟹ better ⇓

```
(def (fac n)
     (def (facc n acc)
          (if (< n 1) acc
              (facc (- n 1) (* n acc))))
     (facc n 1))
(def (fib n)
     (def (fibb n a b)
          (if (< n 2) n
              (fibb (- n 1) b (+ a b))))
     (fibb n ∅ 1))
(def (len l)
     (def (lenn l a)
          (if (null? l) a
              (lenn (cdr l) (+ 1 a))))
     (lenn l ∅))
```

HOF

Scheme II ②

```
(def (sum l)
     (if (null? l) ∅
         (+ (car l) (sum (cdr l)))))
```

$\Longrightarrow$ ∴ tailify? (LAETS)

prod?

inner prod?

```
(def (ip l m)  ;; precond (len l) = (len m).   BUG
     (if (null? l) ∅        ;;  Σ_i l_i m_i
         (+ (* (car l) (car m))
            (ip (cdr l) (cdr m)))))
```

$\Longrightarrow$ TAILIFY (LAETS).

repetitive code BORING!!!!

just as boring as `for (i=0; i<n; ++i)`

`for (p=list; p≠null; p=p.link).`

HOF

Scheme II (3)

Higher Order Functions

reduction

$(foldl\ f\ u\ l) \equiv (((u\ f\ l_0)\ f\ l_1)\ f\ l_2)\ f\ l_3 + \cdots$

$(foldr\ f\ u\ l) \equiv l_0\ f\ (l_1\ f\ (l_2\ f\ (\cdots\ (l_{n-1}\ f\ u$

fold: reduce a list to a unit given assoc, fn, ident unit

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

```
(define (sum l)(foldl + ∅ l))
(define  (prod l)(foldl * 1 l))
```

ex: (sum '(1 2 3)) = (foldl + 0 '(1 2 3))
= (foldl + 1 '(2 3))
= (foldl + 3 '(3))
= (foldl + 6 '())
= 6

O(1) stk

```
(define (sumr l) (foldr + ∅ l))
```

(sumr '(1 2 3)) = (foldr + 0 '(1 2 3))
= (+ 1 (foldr + 0 '(2 3))
= (+ 1 (+ 2 (foldr + 0 '(3)
= (+ 1 (+ 2 (+ 3 (foldr + 0 '()

= 6

O(n) stk

HOF tailRec. Scheme II (4)

∴ foldl ~~better~~, but works only if operator is associative

ex: (foldl ∅) ≠ (foldr ∅).

- 0 (1 2 3)
((0 - 1) - 2) - 3)
= 0 - (1 + 2 + 3)
≡ neg sum

- 0 (1 2 3)
= 1 - (2 - (3 - 0))
= 1 - 2 + 3 - 0
≡ alt diff.

(def (len l) (foldl (λ (n _) (+ n 1)) ∅ l)
α

(len '(1 2 3)) = (foldl α ∅ '(1 2 3))
= (foldl α 1 '(2 3))
= (foldl α 2 '(3))
= (foldl α 3 '())
= 3.

~~(define (filter p? l) ;; ∀x∈l |(p? x)~~
~~(cond ((null? l) '())~~
~~(p?~~

grep.

HOF

Scheme II (5)

```
(define (filter p? l) ;; ∀x←l | p? l
  (cond ((null? l) '())
        ((p? (car l)) (cons (car l)
                            (filter p? (cdr l))))
        (else (filter p? (cdr l)))))
```

(VERBOSE!))

```
(define (filter p? l)
  (if (null? l) '()
      (let ((al (car l))
            (fpdl (filter p? (cdr l))))
        (if (p? al) (cons al fpdl)
            fpdl))))
```

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

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

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

must be local to capture p? because f MUST BE BINARY.

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

HtF

Scheme II (6)

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

⟹ (reverse '(1 2 3)) = (rev '(1 2 3) '())
= (rev (2 3) (1))
= (rev (3) (2 1))
= (rev () (3 2 1))
= (3 2 1).

(TAIL REC)
maybe. ∴ foldl ??

(Δ (rcons d a) (cons a d))

(Δ (reverse l) (foldl (λ (d a) (cons a d)) '() l))

ex: (reverse '(1 2 3))
= (foldl α '() '(1 2 3))
= (foldl α '(1) '(2 3))
= (foldl α '(2 1) '(3))
= (foldl α '(3 2 1) '())
= '(3 2 1)

---

tail recursive foldr? — worth while?
— churn heap 2× fast.

~~(define (foldr f z u l)~~
~~(foldl)~~

```
(define (foldr f2 u l)
        (foldl (λ (x y) (f2 y x)) u (reverse l)))
```

HOF

Scheme II

6 1/2

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

O(n) $n = \|l\|$

not n^2

```
(define (append l m)
    (foldr cons m l)))
```

```
(append '(1 2 3) '(4 5 6))
(foldr cons (456) (123))
(cons 1 (foldr cons (456) (23)
(cons 1 (cons 2 (cons 3 (foldr con(456) ()
(cons 1 (cons 2 (cons 3 (456))
(1 2 3 456)
```

what if fn has no identity?

max, min

```
(define (min l)
        (foldl (λ(x y)(if(< x y) x y)) (car l)(cdr l)))
```

⇒ crashes if null arg.

```
(define (min default l)
        (foldl (λ(x y)(if (< x y) x y)) default l)))
```

~~(define (min default) l)~~
~~(foldl (λ(x y)(if(< x y) x y)) (car f)~~

MAP

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

```
(map1 (λ(x)(* x 2)) '(1 2 3))
      ⇒ (2 4 6).
```

```
(define (map2 f l1 l2)
        (if (null? l1) '()
            (cons (f (car l1)(car l2))
                  (map2 f (cdr l1)(cdr l2)))))
```

(!!) ⇒ error if (len l1) ≠ (len l2)

$ip(v,w) = \sum_i v_i w_i$

```
(define (ip l1 l2)
        (fold l + 0
              (map2 * l1 l2)))
```

```
(define (map1 f l)
        (foldr (λ(a d)(cons (f a) d)) '() l))
                α                       ↑ f is lexical scope.
```

```
(map1 sqrt '(4 16 100))
   = (foldr α '() '(4 16 100))
   = (α 4 (foldr α '() '(16 100)))
   = (α 4 (α 16 (α 100 '())))
   = (α 4 (α 16 '(10)))
   = (α 4 '(4 10))
   = '(2 4 10).
```

map2 --- oops!?!? --- foldr2?!

```
(define (transpose2 l1 l2)
        (if (null? l1) '()
            (cons (list (car l1)(car l2))
                  (transpose2 (cdr l1)(cdr l2)))))

(transpose2 '(1 2 3) '(4 5 6))
        = '((1 4)(2 5)(3 6))

(define (map2 f l1 l2)
        (map1 (λ(x)(apply f x)) (transpose2 l1 l2)))
```

HOF

Scheme II (9)

(map1 f (map1 g l))

↖ conses a list & discards it O(n) heap waste

better:

(map1 (compose f g) l)

(Δ (compose f g)
   (λ (x) (f (g x))))

---

map2, mgp3 .... ??

```
(define (transpose . listo)
  (if (null? listo) '()
      (cons (map1 car listo)
            (transpose (map1 cdr listo)))))
```

≡ ~~(define (transpose . listo)~~
~~(foldr (λ (a d) (cons (caa a) d)) '() listo))~~

(~~transpose~~ '((1 2 3)(4 5 6)(7 8 9)))

= ~~(foldr α '() ((1 2 3)(4 5 6)(7 8 9))~~

~~= (α (1 2 3) (foldr α '() (4 5 6)(7 8 9)))~~

~~= (α (1 2 3) (α (4 5 6) (α (7 8 9) '())))~~

~~= ((α (1 2 3) (α (4 5 6) (α (7 8 9) '())))~~

=

=

HOF

Scheme II (10)

```
(define (transpose . lists)
  (foldr (λ(a d)(map1 car lists)) '() lists))
                                  α
```

ex: (transpose '(1 2 3) '(4 5 6) '(7 8 9))

= (foldr α '() ((1 2 3)(4 5 6)(7 8 9))

~~(α (1 2 3)~~

= (α

```
(define (mapf alist . dlist)
  (let ((t (transpose (cons alist dlist))))
    (map1 apply f t)))
```

~~(define (qsort <? l)~~
~~(let* ((a (car l))~~
~~(lt (grep (λ(x)(<? x a))~~

```
(define (qsort <? l)
  (let* ((a (car l))
         (dl (cdr l))
         (l< (grep (λ(x)(<? x a)) dl))
         (l>= (grep (λ(x)(not(<? x a))) dl)))
    (append (qsort l<) (list a) (qsort l>=))))
```

eval
apply
string<

Warning: Bad Pivot.

Core Scheme grammar

```
prog    → form*
form    → defn | expr
defn    → vardef | (begin defn*)
vardef  → (define var expr)
expr    → const | var
        | (quote datum)
        | (λ formal expr expr*)
        | (if expr expr expr)
        | (set! var expr)
        | applic
const   → bool | num | char | string
formal  → var | (var*)
        | (var var* . var)
applic  → (expr expr*)
```