(Dybvig)

Scheme

- strings, lists, vectors, numbers
- uses: editors, compilers, OS, graphics
- very simple syntax:

  prog → sexpr*

  sexpr → atom | (sexpr*)
- Revised Report R^5RS: 50 pages
- all objs 1st class, including functions
- call by value (applicative order) [eager]
- lexically scoped
- mandated support tail calls
- no loops → recursion
- garbage collected
- programs represented as lists
- unnamed fns called lambdas
- descended of Lisp, ∴ of λ-calculus

1.1 Syntax

identifiers [a-z][A-Z] (case not distinct)
[0-9] ?! . + - * / < = > : $ % ^ & _ ~ @
except must not look like numbers.

strings: " "

comments start w/ ;

' ` , . ( ) [ ] { } are all parens

booleans #t #f

only #f is false

all other exprs true

expressions: prefix notation
(* (- x 2) y)
chars #\a #\space

Naming conventions
predicates end in ?
eq? zero? null?
also = < > <= >=
type predicates: pair? string?
object functions: string-append.
conversions: vector->list
side effects use!
set! vector-set!

2. Interacting
REPL - read eval print loop
top level interactive
mzscheme
```
> 1
1
> (+ 2 3)
5
> '(1 2 3)
(1 2 3)
> (1 2 3)
error: 1 is not a function
```

Data Types

Scheme ③

boolean #t #f [ONLY #f is false]

(boolean? x) (not x)

numbers

integer ⊂ rational ⊂ real ⊂ complex ⊂ number

42 22/7 3.14159 2+3i

each predicate ex (integer? x)

#b1100 = #o14 = #xC = #d12 = 12

predicates = < > <= >=

arith + - * / expt

max min abs

(- 5 2 1) ⇒ 2

(/ 22 7) ⇒ 22/7

(expt 4 1/2) ⇒ 2.0

(+ 1 2 3 4 5) => 15

characters

#\c #\newline #\space

char? char=? char<?

char-ci<? (case insensitive)

symbols

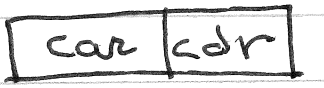
- self evaluating (symbol? x)
- constants

(QUOTE E) 'E

↑ special form.

Dotted Pairs & Lists

IBM 704 CPU:
car = contents of address register
cdr = contents of decrement register

cons cell 

(cons 1 2) 

'(1 . 2)

'(1 2 3) 

'() empty list

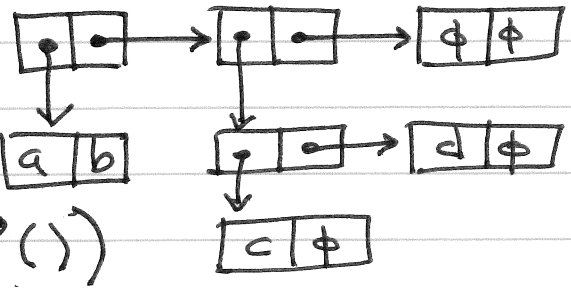
```
(define x '(1 2 3))
  (car x) ⇒ 1
  (cdr x) ⇒ (2 3)
  (cadr x) ⇒ 2      (caar x)} errors
  (cddr x) ⇒ (3)    (cdar x)}
'(a.(b.(c.())) ⇒ (a b c)
```

PROPER Lists linked by cdr-chains
end in '()

predicates: (pair? x) - O(1)
(null? x) - O(1)
(list? x) - O(n)

(list 'a 'b 'c) ⇒ (a b c)
(list a b c) ⇒ (20 30 40)

picture



((a . b)((c) d)'())

2.3 Evaluating Exprs

(fn arg arg...)

1. evaluate fn
2. eval args
2. apply fn to args

BUT: (Quote) does not evaluate
it is special form

2.4 Variables; Let-Exprs

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

Syntax
(let ((var val) ...) exp exp...)
--last expr is value

functions are just values

(let ((f +)) (f 2 3)) ⇒ 5

(let ((f +) (x 2) (y 3))
  (f x y)) ⇒ 5 — local defn

(let ((+ *)) (+ 2 3)) ⇒ 6
(+ 2 3) ⇒ 5

2.5 Lambda Exprs

(lambda (x y) (+ x y))
args, body

(λ (arg ...) exp exp ...)

((λ (x) (+ x x)) 4) ⇒ 8

λ-expr is an object like any other

((λ (x) (+ x x)) (* 3 4)) ⇒ 24
fn, arg

Free variable, bound variable

(λ (x) (+ x y))

let is just a λ-expr

(let ((x a)) (cons x x))
⇒ ((λ (x) (cons x x)) a)

Hygienic MACRO

Varargs

(λ (x y . z) ξ ξ ...)

↑ ↑ ↑ list of rest

arg1 arg2

2.6 Top Level Definitions

```
(define x 6)
(define d (lambda (f x)(f x x)))
  (d + 10) => 20

(define cadr (λ (x) (car (cdr x))))
```

abbreviations

```
(define (cadr x)(car (cdr x)))
```

2.7 Conditionals

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

(define abs (λ(n)
      ((if (< n 0) - +) 0 n)))

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

(if) is a special form (lazy)

(not x) =>

(or e_1 e_2 e_3 ....) =>

returns first arg not false
returns #f if all false

(and e_1 e_2 e_3 ...) =>

returns #t if all true; #f if any false

Predicates: = < > <= >=
on numbers

(null? x) on lists

eq? — identity
eqv? — equivalent
equal? — structurally equivalent

pair?
symbol?
number?
string?
} name of type? asks if it is

```
(define (sign n) (if (< n 0) -1
                     (if (> n 0) +1
                         0)))
```
} Nested

```
(define (sign n)
        (cond ((< n 0) -1)
              ((> n 0) +1)
              (else 0)))
```

2.8 Simple Recursion

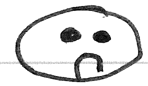

```
(define len (lambda (l)
        (if (null? l)
            0
            (+ (len (cdr l)) 1))))
```

⇒ BAD ⇒ O(n) recursion on len of list

☆ accumulator style tail call

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

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



| O(n) stack
| $O(n^2)$ time
| $O(n^2)$ heap

accumulator:

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

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

↑ two stacks

~~(define (roots a b c)~~ Scheme (10)

ex: roots of $ax^2+bx+c=0$

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

eliminate common subexprs

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

Mapping

```
(define (abslst l)
        (if (null? l) '()
            (cons (abs (car l))
                  (abslist (cdr l)))))
(define (abslist l) (map abs l))
    ;; like a for(i:s) in Java/C++
(map (λ(x)(* x x)) '(1 2 3)) => (1 4 9)
(define (map f l)
        (if (null? l) '()
            (cons (f (car l))
                  (map f (cdr l)))))
```

3.2 More Recursion


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

O(n) stack

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

$O(2^n)$ time

O(n) stack

TAIL RECUR

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

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

outer fn is interface
inner fn is worker

Let

Scheme (12)

(let → exprs eval outside of let

(let* → exprs in sequence
vars avail in next

(letrec → allows mutual recursion

assume global x = 20

(let ((x 1)(y x))(+ x y))
⇒ 21

(let* ((x 1)(y x))(+ x y)) ⇒ 2
because y is the local x

5.1 Procedure Apply

(f a a ...) applies f to args a a ...

(apply + '(4 5)) ⇒ 9

(begin e_1 e_2 e_3 ...) ⇒ result of last expr

5.3 Conditionals

(if test consequent alternate)

(not x) ≡ (λ(x)(if x #f #t))

(and e_1 e_2 ...) short circuit
evals args: if any #f returns #f
else returns last
(and) => #t

(or e_1 e_2 ...) => first not #f
(or) => #f

(cond clause...)
each clause (test)
(test ex_1 ex_2 ...)
(test => ex)
(else ex_1 ...) ← last clause only

5.4 Recursions & Mapping

(map f l_1 l_2 l_3 ...)
lists must be same length
f must accept #args as lists
returns a single list

(map + '(1 2 3) '(4 5 6) '(7 8 9))
=> (12 15 18)

(for-each f l_1 l_2 ...)
as for map but does not return a list; used for effect

6 Operations on Objects

~~quote~~ 'x quote

`x quasiquote

,x unquote

quasiquote allows unquoted portions

`(+ 2 ,(* 3 4)) => (+ 2 12)

(eq? x y) – #t if objs identical
– can't compare numbers & chars

(eqv? x y) – equivalence
– can compare nums & chars

(equal? x y) – same structure
– expensive on lists & vectors

others

boolean?	number?	char?
null?	complex?	string?
pair?	real?	vector?
list?	rational?	symbol?
	integer?	

6.3 Lists & Pairs

cons makes a pair (cell)

proper lists linked by cdr ends with '()

car, cdr extracts

caar	caaar	cdaar	up to 4
cadr	caadr	cddar	
cdar	cadar	cdadr	c[ad]{1,4}r
cddr	caddr	cdddr	

(list ...) returns a list of its args.

(length l)

(append l_1 l_2 l_3 ...)

(reverse l)

(member obj list)

6.4 Numbers

integer ⊂ rational ⊂ real ⊂ complex ⊂ number

note: 3/4 is a rational / is not a div symbol

$$\begin{pmatrix} = \\ < \\ > \\ <= \\ >= \end{pmatrix} n_1\ n_2\ n_3 \cdots$$ monotonic

(+ num...)
(-)
(*)
(/)
} one or many args

(quotient i j)
(remainder i j) ← sign of numerator
(modulo i j) ← sign of denominator

--- lots of math fns
--- look them up in R^5RS

6.6 Strings

(string=? many)
(string<? etc...

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*)
```