Scheme in Scheme

Why implement Scheme in Scheme

- Implementing a language is a good way to learn more about programming languages
- Interpreters are easier to implement than compilers, in genera
- Scheme is a simple language, but also a powerful one
- Implementing it first in Scheme allows us to put off some of the more complex lower-level parts, like parsing and data structures
- While focusing on higher-level aspects

Lisp and Scheme are simple

- Simple syntax and semantics
- John McCarthy's original Lisp had very little structure:
  - Procedures CONS, CAR, CDR, EQ and ATOM
  - Special forms QUOTE, COND, SET and LAMBDA
  - Values T and NIL
- The rest of Lisp can be built on this foundation (more or less)

Meta-circular Evaluator

- We'll look at an adaptation of the metacircular evaluator for Scheme from Abelson and Sussman's book, the Structure and Interpretation of Computer Programs (SICP)

- "A meta-circular evaluator is a special case of a self-interpreter in which the existing facilities of the parent interpreter are directly applied to the source code being interpreted, without any need for additional implementation. Meta-circular evaluation is most common in the context of *homoiconic languages*".

Meta-circular Evaluator

- Homoiconicity is a property of some programming languages

- From *homo* meaning the same and *icon* meaning representation

- A programming language is homoiconic if its primary representation for programs is also a data structure in a primitive type of the language itself

Meta-circular Evaluator

- We'll not do all of Scheme, just enough for you to understand the approach

- We can us the same approach when writing an interpreter for Scheme in Python

- Since the MCE requires *mutable pairs*, we'll run it in the R5RS dialect
  - I'm still working on a version that can run in the latest PLT scheme

Mutable Pairs?

- Scheme calls a cons cell a pair
- Lisp always had special functions to change (aka destructively modify or mutate) the components of a simple cons cell
- Can you detect a sentiment there?
- RPLACA (RePLAce CAr) was Lisp's function to replace the car of a cons cell with a new pointer
- RPLACD (RePLAce CDr) clobbered the cons cell's cdr pointer
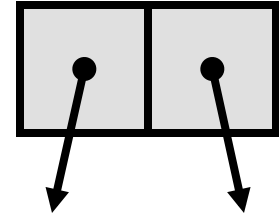
Lisp's rplaca and rplacd

```
GL% clisp
...
[1]> (setq l1 '(a b))
(A B)
[2]> (setq l2 l1)
(A B)
[3]> (rplaca l2 'foo)
(FOO B)
[4]> l1
(FOO B)

[5]> l2
(FOO B)
[6]> (rplacd l1 '(2 3 4))
(FOO 2 3 4)
[7]> l1
(FOO 2 3 4)
[8]> l2
(FOO 2 3 4)
```

Scheme's set-car! & set-cdr!

> (define l1 '(a b c d))

> l1

(a b c d)

> (set-car! l1 'foo)

> l1

(foo b c d)

> (set-cdr! l1 '(2 3))

> l1

(foo 2 3)

> (set-cdr! l1 l1)

> l1

#0=(foo . #0#)

> (cadr l1)

foo

> (caddr l1)

foo

> (cadddr l1)

foo

kicked out of R6RS

- Scheme removed set-car! and set-cdr! from the language as of R6RS
  - They played to their ideological base here
  - Or maybe just eating their own dog food
- **R6RS** is the Revised **6 Report on the Algorithmic Language Scheme
- R6RS does have a library, mutable-pairs, provides a new datatype for a mutable pair and functions for it
  - mcons, mcar, mcdr, mlist, …set-mcar!, set-mcdr!

aside: PL standards

- Some languages are created/promoted by a company (e.g., Sun:Java, Microsoft:F#, Apple:Objective C)

- But for a language to really be accepted, it should be defined and maintained by the community

- And backed by a well-defined standard

- That may be supported by a recognized standards organizations (e.g., IEEE, ANSI, W3C, etc)

RnRS

- Scheme is standardized in the official IEEE standard and via a de facto standard called the *Revised^n Report on the Algorithmic Language Scheme*
- Or RnRS
- Common versions:
  - R5RS in 1998
  - R6RS in 2007

mutable-pairs

> (define l1 (cons 1 (cons 2 empty)))

> l1

(1 2)

> (define m1 (mcons 1 (mcons 2 empty)))

> m1

{1 2}

> (car l1)

1

> (car m1)

. . car: expects argument of type <pair>; given {1 2}

> (mcar m1)

1

> (set-car! l1 'foo)

. . reference to undefined identifier: set-car!

> (set-mcar! l1 'foo)

. . set-mcar!: expects type <mutable-pair> as 1st argument, given: (1 2); other arguments were: foo

> (set-mcar! m1 'foo)

> m1

{foo 2}

How to evaluate an expression

- We'll sketch out some rules to use in evaluating an s-expession

- Then realize them in Scheme

- The only tricky part is representing an environment: binding symbols to values

  - Environments inherit from other environments, so we'll consider an environment to be a set of frames

  - We'll start with a global environment

Environment

- An environment is just a list of frames
- The first frame is the current environment, the second is the one it inherits from, the third is the one the second inherits from, etc.
- The last frame is the global or top level environment

Eval an Atom

- **Self-Evaluating** - Just return their value

  - Numbers and strings are self evaluating

- **Symbol** - Lookup closest binding and return

  - Raise an error if not found

Eval a "special form"

Special forms are those that get evaluated in a special, non-standard way

- (quote X) – return X

- (define X B) – bind X to evaluation of B

- (lambda VARS BODY) - Make a procedure, write down VARS and BODY, do not evaluate

- (set! X Y) – find X binding name, eval Y and set X to the return value

- (if X Y Z) – eval X and then eval either Y or Z

Eval a procedure call

- **Primitive: (F . ARGS)**

  - Apply by magic...

- **User-defined: (F . ARGS)**

  - Make a new *environment frame*

  - Extend to procedures frame

  - Bind arguments to formal parameters

  - Evaluate procedure body in the new frame

  - Return its value

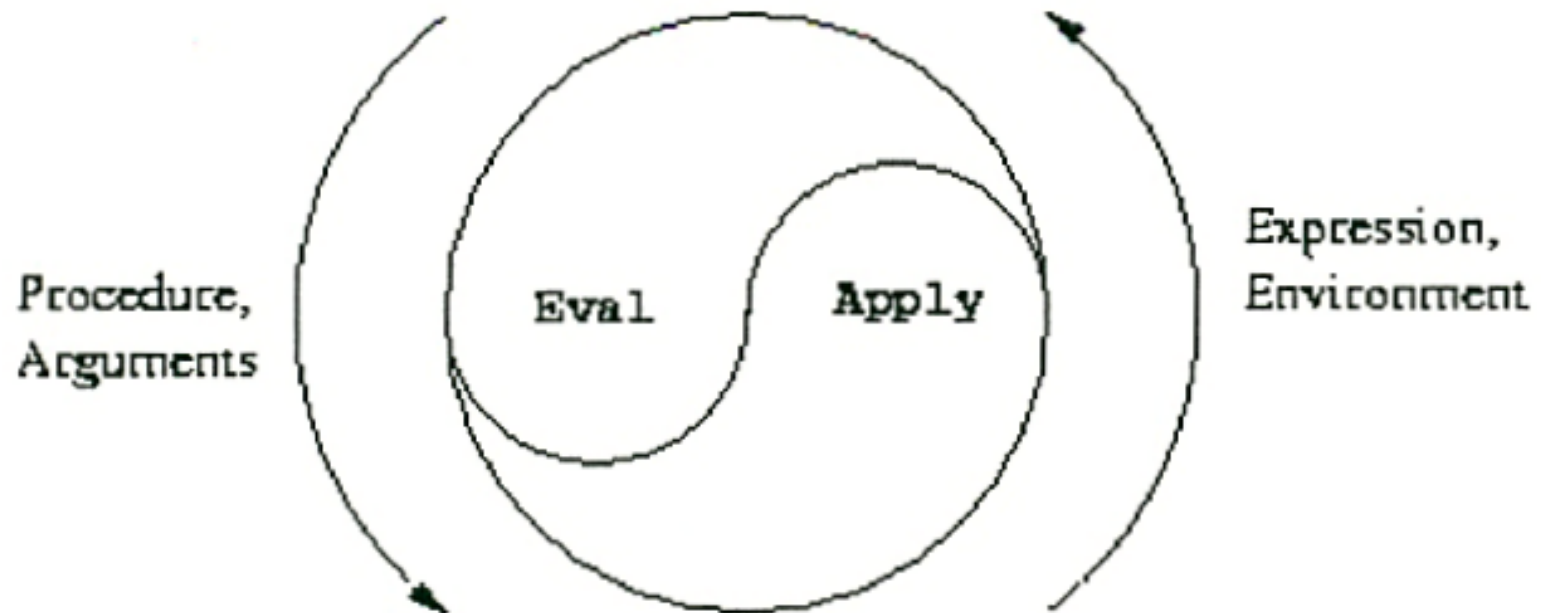# Scheme in Scheme

```scheme
(define (scheme)

  (print '|> |)

  (print (mceval

            (read)

            the-global-environment))

  (scheme) )
```

The Yin and Yang of Lisp

Eval

```scheme
(define (mceval exp env)
  (cond ((self-evaluating? exp) exp)
        ((symbol? exp) (lookup exp env))
        ((special-form? exp)
         (do-something-special exp env))
        (else (mcapply (mceval (car exp) env)
                       (map (lambda (e) (mceval e env))
                            (cdr exp)))) ) ) )
```

apply

```
(define (apply op args)
  (if (primitive? op)
      (do-magic op args)
      (mceval (op-body op)
              (extend-environment
                (op-formals op)
                args
                (op-env op)))))
```

What's in a function?

- In Scheme or Lisp, the representation of a function has three parts:
  - A list of the names of its **formal parameters**
  - The expression(s) that make up the function's **body**, i.e. the code to be evaluated
  - The **environment** in which the function was defined, so values of non-local symbols can be looked up
- We might just represent a function as a list like
  (procedure (x y) (+ (* 2 x) y) (… env …))

What's an environment

- An environment is just a list of environment **frames**
  - The last frame in the list is the global one
  - The nth frame in the list extends the n+1th
- An environment frame records two things
  - A list of **variables** bound in the environment
  - The **values** they are bound to
- Suppose we want to extend the global environment with a new local one where x=1 and y=2

Environment example

- Consider entering:
  - ➢ (define foo 100)
  - ➢ (define (square x) (* x x))
- The environment after evaluating the first two expressions would look like:

(((square foo …) (procedure (x)(* x x)) 100 …)
      variable names   variable values
)

Environment example

- Consider entering:
  - ➤ (square foo)

- Scheme evaluates *square* and *foo* in the current environment and pushes a new frame onto the environment in which x is bound to 100

```
(   (( x ) 100 )
    ((square foo …)  (procedure (x)(* x x ))  100 …)
)
```

Running the MCE…

```scheme
(define (scheme)
  (display "Type `exit' to leave MCE\n")
  (schemeloop))


(define (schemeloop)
  (display "\nMCE> ")
  (let ((input (read)))
    (if (equal? input 'exit)
      'done
      (begin (display (mceval input the-global-environment))
             (schemeloop)))))
```

mceval

```
(define (mceval exp env)
  (cond ((self-evaluating? exp) exp)
      ((symbol? exp) (lookup-variable-value exp env))
      ((quoted? exp) (cadr exp))
      ((assignment? exp) (eval-assignment exp env))
      ((definition? exp) (eval-definition exp env))
      ((if? exp) (eval-if exp env))
      ((lambda? Exp)
        (make-procedure (cadr exp) (cddr exp) env))
      ((begin? exp) (eval-sequence (cdr exp) env))
      ((application? exp)
       (mcapply (mceval (car exp) env)
          (map (lambda (x)(mceval x env)) (cdr exp))))
      (else (error "mceval: Unknown expression type" exp))))
```

mcapply

```
(define (mcapply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
        ((defined-procedure? procedure)
         (eval-sequence
           (proc-body procedure)
           (extend-environment
             (proc-parameters procedure)
             arguments
             (proc-environment procedure))))
        (else (error "mceval: Unknown proc. type" procedure))))
```

Global Environment

```
(define the-empty-environment '())

(define (setup-environment)
 (let ((initial-env
        (extend-environment primitive-proc-names
                            primitive-proc-objects
                            the-empty-environment)))
   (define-variable! 'empty '() initial-env)
   initial-env))

(define the-global-environment (setup-environment))
```

Extending an environment

```
(define (extend-environment vars vals base-env)
  (if (= (length vars) (length vals))
      (cons (make-frame vars vals) base-env)
      (if (< (length vars) (length vals))
          (error "Too many arguments supplied" vars vals)
          (error "Too few arguments supplied" vars vals))))

(define (make-frame variables values)
    (cons variables values))
```

```
(define (lookup-variable-value var env)
  (define (env-loop env)

    (define (scan vars vals)
     (cond ((null? vars)
              (env-loop (enclosing-environment env)))
            ((eq? var (car vars)) (car vals))
            (else (scan (cdr vars) (cdr vals)))))

    (if (eq? env the-empty-environment)
       (error "Unbound variable" var)
       (let ((frame (car-frame env)))
          (scan (frame-variables frame)
               (frame-values frame)))))
  (env-loop env))
```

```
(define (define-variable! var val env)
  (let ((frame (car-frame env)))
    (define (scan vars vals)
      (cond ((null? vars)
              (add-binding-to-frame! var val frame))
            ((eq? var (car vars))
              (set-car! vals val))
            (else (scan (cdr vars) (cdr vals)))))
    (scan (frame-variables frame)
          (frame-values frame))))
```

```
(define (set-variable-value! var val env)
  (define (env-loop env)
    (define (scan vars vals)
      (cond ((null? vars) (env-loop (enclosing-environment env)))
            ((eq? var (car vars))  (set-car! vals val))
            (else (scan (cdr vars) (cdr vals)))))
    (if (eq? env the-empty-environment)
        (error "Unbound variable -- SET!" var)
        (let ((frame (car-frame env)))
          (scan (frame-variables frame)  (frame-values frame)))))
  (env-loop env))
```