Scheme in Scheme 1

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 general
- 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

- "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*".
- We'll look at an adaptation from

  Abelson and Sussman, Structure and Interpretation of Computer Programs, MIT Press, 1996.

Meta-circular Evaluator

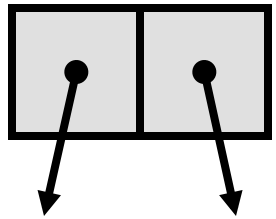
- Homoiconicity is a property of some programming languages
- From *homo* meaning the same and *icon* meaning representation
- A programming language is homoiconic if its primary representation for programs is also a data structure in a primitive type of the language itself
- Few examples: Lisp, Prolog, Snobol

Meta-circular Evaluator

- We'll not do all of Scheme, just enough for you to understand the approach
- We can us the same approach for an interpreter for Scheme in Python
- To provide reasonable efficiency, we'll use mutable-pairs

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

Frame

- An environment frame is just an (unordered) collection of bindings
- A binding has two elements: a symbol representing a variable and an object representing its (current) value
- An environment might be represented as

```
( ( (x 100) (y 200) )
  ( (a 1) (b 2) (x 2) )
  ( (null '()) (empty '()) (cons …) …)
)
```

Eval an Atom

- **Self-Evaluating** - Just return their value
  - Numbers and strings are self evaluating
- **Symbol** - Lookup closest *binding* in the current environment and return its second element
  - 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

Our strategy

- We’re implementing an interpreter for Scheme in Scheme
- The host language (Scheme) will do many details: data representation, reading, printing, primitives (e.g., cons, car, +)
- Our implementation will focus on a few key parts: eval, apply, variables and environments, user defined functions, etc.

McScheme interpreter in Scheme

http://cs.umbc.edu/331/f11/code/scheme/mcs/

- **mcs.ss**: simple Scheme subset
- **mcs_scope.ss**: larger Scheme subset
- **mcs_basics.ss**: 'library' of basic functions
- **readme.txt**: short intro text
- **session.txt**: example of McScheme in use

Limitations

- define can only assign a variable to a value, i.e., 1st arg must be a symbol. Define functions like:

  (define add1 (lambda (x) (+ x 1))

- lambda only allows one expression in body; for more use begin:

  (lambda (x) (begin (define y (* x x)) (* y y)))

- No set! to assign variables outside of the local environment (e.g., global variables)

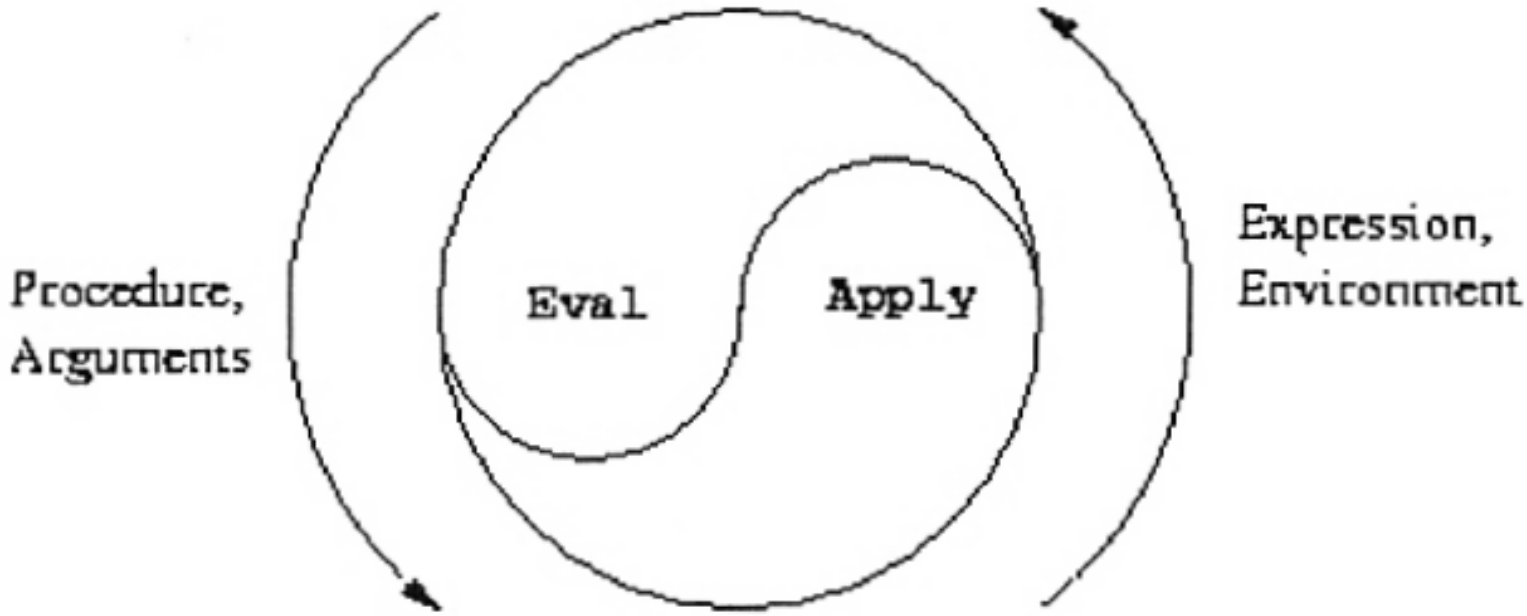
REPL

Here's a trivial read-eval-print loop:

```
(define (mcscheme)
   ;; mcscheme read-eval-print loop
   (printf "mcscheme> ")
   (mcprint (mceval (read) global-env))
   (mcscheme))

(define (mcprint x)
   ;; Top-level print: print x iff it's not void
   (or (void? x) (printf "~s~n" x)))
```

The Yin and Yang of Lisp



The eval and apply operations have been fundamental from the start

Simple mceval

```
(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))) ) ) )
```

mcapply

```
(define (mcapply 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 (lambda (x) (* x x)))
  - (define x -100)
- The environment after evaluating the first three expressions would look like:

```
(  ( (x . -100)
     (square lambda (x)(* x x ) #0)
     (foo . 100)
     …)
)
```

Environment example

- Consider entering:
  - (square foo)
- mcscheme 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 ))
    ((x . -100)
      (square lsmbda (x)(* x x ))
      (foo . 100) ... )
)
```

Lets look at the code

Take a look at the handout

mceval

```
(define (mceval exp env)
 (cond
   ((or (number? exp) (string? exp)
        (boolean? exp) (eof-object? exp)) exp)
   ((symbol? exp) (lookup exp env))
   ((eq? (first exp) 'quote) (second exp))
   ((eq? (first exp) 'begin) (last (map (lambda (x)(mceval x env)) (rest exp))))
   ((eq? (first exp) 'if)  (if (mceval (second exp) env)
                               (mceval (third exp) env)
                               (mceval (fourth exp) env)))
   ((eq? (first exp) 'define)
    (mcdefine (second exp) (mceval (third exp) env) env))
   ((eq? (first exp) 'load) (call-with-input-file (second exp) mcload))
   ((eq? (first exp) 'lambda)
    (list 'LAMBDA (second exp) (third exp) env))
   (else (mcapply (mceval (first exp) env)
                  (map (lambda (x)(mceval x env)) (rest exp))))))
```

mcapply

```
(define (mcapply proc args)
 ;; apply procedure proc to arguments args
 (cond ((procedure? proc) (apply proc args))
        ((and (pair? proc) (eq? (first proc) 'LAMBDA))
         (mceval (third proc)
                  (cons (make-frame (second proc) args)
                  (fourth proc))))
       (else
         (mcerror "mcapply: Undefined procedure" proc))))
```

Global Environment

```
(define (make-frame vars values)
   ;; Makes an environment frame with variables
   ;; vars and initial values values, e.g.
   (mmap mcons (l2ml vars) (l2ml values)))

(define (l2ml l)
   ;; takes a list and returns a mutable list (mlist)
   (if (null? l) l (mcons (car l) (l2ml (cdr l)))))
```

```
> (make-frame '(a b) '(1 2))
{ {a . 1} {b . 2} }
```

Global Environment

```
;;  Primitives defined as their Scheme counterparts
(define builtins '(car cdr cons number? pair?
                   string? eq? + - * / = < > print eof))

;; intial global environment
(define global-env
  (list (make-frame builtins (map eval builtins))))
```

```
> global-env
({ {car . #<procedure:car>}
   {cdr . #<procedure:cdr>}
    ...
   {eof . #<eof>} } )
```

Looking up a value

```
(define (lookup var env)
 ;; return value of variable var in environment env
 (cond ((null? env) (mcerror "unbound: " var))
          ((massoc var (first env))
           (mcdr (massoc var (first env))))
         (else (lookup var (rest env)))))
```

```
; massoc = assoc for mutable-pairs.  Returns
; tuple in 2nd arg whose car equals 1st arg
> (massoc 'cons (car global-env))
{cons . #<procedure:cons>}
```

Defining a variable

```
(define (mcdefine var val env)
 ;; define var in environment env, giving it value val
 (let ((frame (first env)))
    (if (massoc var frame)
        ;; variable already defined, change it's value
        (set-mcdr! (massoc var frame) val)
        ;; add a new var-val cell to the end of the frame
        (set-mcdr! (mlast-pair frame)
                   (mcons (mcons var val) null))))
  (void))
```

Defining a variable

```
> (define e (list (mlist (mlist 'a 1) (mlist 'b 2))))
> e
({{a 1} {b 2}})
> (mcdefine 'b -2 e)
> e
({{a 1} {b . -2}})
> (mcdefine 'c 3 e)
> e
({{a 1} {b . -2} {c . 3}})
>
```

- Define only works on the current frame, i.e., first frame in environment
- If it finds the variable, it changes its value
- Otherwise, it adds a new tuple at the frame's end

Setting a variable

```
(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))
```

A sample session

```
> (load "mcs.ss")
"mcscheme:, (mcscheme) to start, ^C to leave"
> (mcscheme)
mcscheme> 100
100
mcscheme> (+ 100 200)
300
mcscheme> (define fact (lambda (n) (if (< n 2) 1 (* n (fact (- n 1))))))
mcscheme> fact
#0=(LAMBDA (n) (if (< n 2) 1 (* n (fact (- n 1)))) ({{car
 . #<procedure:car>} … {eof . #<eof>} {fact . #0#}}))
mcscheme> (fact 8)
40320
mcscheme> (load "mcs_basics.ss")
mcscheme> (map add1 '(1 2 3 4 5))
(2 3 4 5 6)
mcscheme> (reverse '(1 2 3 4 5))
(5 4 3 2 1)
```

Conclusion

- We studied an interpreter for a very limited subset of Scheme
- It relies on the host language (Scheme) for many details (e.g., representing lists, primitive functions, read and print)
- Key concepts: eval, apply, environments, use-defined functions
- Using this as a base, we can expand the Scheme subset covered
- And use it as a model when implementing Scheme in other languages