Scheme in Scheme?!?

…the metacircular evaluator…

What We've Learned…

- **Implement functional programming**
  The basic operations of a computer that can be used as a building block for further layers of complexity.
- **Create data abstractions**
  Used to simplify our understanding of CS and to invent solutions to problems that a computer can mirror.
- **Implement message passing**
  Used to implement OOP - furthering a grandiose type of ADT, Multiple Independant Intelligent Agents, which can create "living" types.

What We've Learned…

- **Deal with tree data structures**
  Used to implement databases and hierarchtical structures efficiently. Most "real-world" data deal with this ADT.
- **Create infinite data structures**
  Delayed evaluation of data (evaluated as needed) as seen with Streams.

Isn't it amazing what we've done so far???

Next Idea: Evaluators

- So now it's on to evaluators
- So we've been using: **Underlying Scheme**
- Time to create new models of evaluation…WHY?

New Models of Evaluation…

- One: to embody the common elements of large groups of problems.
- Two: To solve problems *differently*, to think outside of the box.
- So outside of the box we'll be going for the next 3 weeks…

Different Evaluators

- The differences (and advantages) of lexical vs. dynamic scope. **(Scheme vs. Logo)**
- A faster compiler/interpreter **(Analyze)**
- A normal-order Scheme evaluator **(Lazy)**
- A version of Scheme that solves problems non-deterministically **(Amb)**
- A pattern-matcher/artificial intelligence Scheme evaluator **(Query)**

So What Now?

- Well since we know **SCHEME** really well RIGHT? ☺
- We're going to write Scheme in Scheme.
- This is called the **metacircular evaluator**

MCE in all it's glory…

- So the environment diagram showed us the **"below the line"** evaluation of scheme expressions
- This is going to come into play right now…so let's review **THE RULES**!

The Rules!

- **Self-Evaluating** - Just return their value
- **Symbol** - Return closest binding, if none error.

…more stuff to follow ☺

More rules…

- **Special forms:**
  Define - bind var name to evaluation of rest in current frame

  Lambda - Make a procedure, write down params, and body - Do not evaluate

  Begin - Evaluate each expression, return value of last one

  set! - find var name, eval expression and set var to the return value

  if - eval predicate and then either the true-part or false-part.

Some more rules…

- **Procedures**
  Primitive's - Apply by magic...

  User-defined - Make a new frame, extend to proc's frame, bind arguments to formal parameters, evaluate the body of the procedure in the new frame.
- ***Syntactic Sugar*** *- Get rid of it (untranslate)!*

What to do…

- We have all the rules to do Scheme.
- Now let's translate it into a Scheme evaluator.
- There's only 2 things we do in Scheme:
  - Evaluate expressions
  - Apply operator to arguments in a new environment

Eval (from reader)

```
(define (scheme)
      (print '|> |)
      (print (eval (read) the-global-environment))
      (scheme) )

(define (eval exp env)
      (cond ((self-evaluating? exp) exp) ;;Rule 1
              ((symbol? exp) (lookup exp env)) ;;Rule 2
              ((special-form? exp)
               (do-something-special exp env)) ;;Rule 3
             (else (apply (eval (car exp) env) ;;Rule 4
                         (map (lambda (e) (eval e env))
                               (cdr exp))) ) ) )
```

Apply (from reader)

```
(define (apply op args) ;;Rule 4... Verbatim
     (if (primitive? op)
         (do-magic op args)
         (eval (body op)
                (extend-environment
                         (formals op)
                         args
                         (op-env op) ))))
```

So Far…

- That’s what we had so far in class and the reader…but what about the book???
- Chapter 4 shows you a detailed way to do the mce
- Let’s take a further look…

Running the MCE…

So here’s what you run for the MCE:

```
(define (mce)
     (set! the-global-environment
             (setup-environment))
     (driver-loop))
```

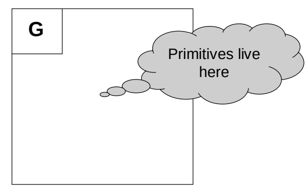
What’s it doing?

Global Environment

- So it sets up the global environment.
  - At first the global environment is defined as: (define the-global-environment '())
  - But in mce it gets set! to be (setup-environment)
- Now what's happening there?

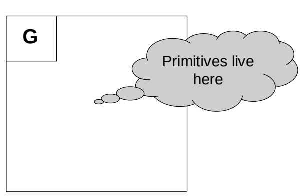
Environments…

- First off, what's an environment?
  - **Place to store variable bindings**



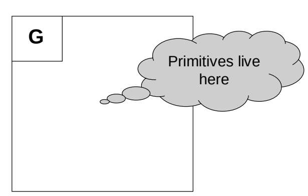
Environments…

- First off, what's an environment?
  - Place to store variable bindings
  - **Where procedures point to**



Environments…

- First off, what's an environment?
  - Place to store variable bindings
  - Where procedures point to
  - **Can be extended by a frame upon procedure invocations**



Setting up the environment…

```
(define (setup-environment)
    (let ((initial-env
            (extend-environment (primitive-procedure-names)
                                (primitive-procedure-objects)
                                the-empty-environment)))
        (define-variable! 'true true initial-env)
        (define-variable! 'false false initial-env)
        (define-variable! 'import
                (list 'primitive
                    (lambda (name)
                        (define-variable! Name
                                    (list 'primitive (eval name))
                                    the-global-environment)))
                    initial-env)
        initial-env))
```

Creating a global environment in the MCE…

- What do we first start with?
  - the-empty-environment: (define the-empty-environment '())
- So the global environment starts off as empty list with no variable bindings.

Setting up the environment…

```
(define (setup-environment)
    (let ((initial-env
            (extend-environment (primitive-procedure-names)
                                (primitive-procedure-objects)
                                the-empty-environment)))
        (define-variable! 'true true initial-env)
        (define-variable! 'false false initial-env)
        (define-variable! 'import
                (list 'primitive
                    (lambda (name)
                        (define-variable! Name
                                    (list 'primitive (eval name))
                                    the-global-environment)))
                    initial-env)
        initial-env))
```

Primitives…

```
(define primitive-procedures
    (list (list 'car car)
          (list 'cdr cdr)
          (list 'cons cons)
          (list 'null? null?)
          (list '+ +)
          (list '- -)
          (list '* *)
          (list '/ /)
          (list '= =)
          (list 'list list)
          (list 'append append)
          (list 'equal? equal?)
          ;;    more primitives ) )
```

```
(define (primitive-procedure names)
    (map car
         primitive-procedures))

(define (primitive-procedure-objects)
    (map (lambda (proc)
             (list 'primitive (cadr proc)))
         primitive-procedures))

(define (primitive-procedure? proc)
    (tagged-list? proc 'primitive))

(define (primitive-implementation proc)
    (cadr proc))
```

Extending an environment…

- How do we extend 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))))
```

- Make a frame???
  - Let's do it…

Making Frames…

- Frames hold the variables and values.
  - Both are lists.

```
(define (make-frame variables values)
      (cons variables values))
      Ex. ((x y z) 1 2 3)

(define (frame-variables frame) (car frame))
      Ex. (frame-variables ((x y z) 1 2 3)) → (x y z)

(define (frame-values frame) (cdr frame))
      Ex. (frame-values ((x y z) 1 2 3)) → (1 2 3)

(define (add-binding-to-frame! var val frame)
      (set-car! frame (cons var (car frame)))
      (set-cdr! frame (cons val (cdr frame))))
```

Setting up the environment…

So the global environment should look something like this…

(**(***(car cdr cons null? + …)*
(primitive #[closure car])
*(primitive #[closure cdr]) …***)**)

Setting up the environment…

```
(define (setup-environment)
   (let ((initial-env
            '( ((car cdr cons null? + …) (primitive #[closure car]) …) )
      (define-variable! 'true true initial-env)
      (define-variable! 'false false initial-env)
      (define-variable! 'import
            (list 'primitive
               (lambda (name)
                  (define-variable! Name
                           (list 'primitive (eval name))
                           the-global-environment)))
               initial-env)
      initial-env))
```

Defining variables…

Searches in the current frame for the variable, if not it just adds it to the frame, otherwise it changes the value of the variable.

```
(define (define-variable! var val env)
   (let ((frame (first-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))))
```

Global is set…

- So the global environment is set…
- We went through a lot of the environment and frame code of the mce…what happens when we run it?

```
(define (mce)
   (set! the-global-environment
          (setup-environment))
   (driver-loop))
```

- It calls the driver-loop…

Read/Eval/Print Loop

- Driver loop is also called the read/eval/print loop
- Reads in from the user…that's why mc-eval takes a quoted expression to evaluate ie. (mc-eval '(+ 1 2))

Clarification

- Remember when I said:
  "All Scheme Expressions are just LISTS"
- Here is where that comes into play
  > (define x 14)
  this just says that this is a list with the car being a define
- So you could think about this as a tagged object, so tagged-data come into the picture…

Tagged List..

- So here's the general procedure for tagged-list.

```
(define (tagged-list? exp tag)
    (if (pair? exp)
        (eq? (car exp) tag)
        false))
```

Driver Loop…

```
(define (driver-loop)
  (prompt-for-input input-prompt)
  (let ((input (read)))
    (let ((output (mc-eval input the-global-
  environment)))
      (announce-output output-prompt)
      (user-print output)))
  (driver-loop))
```

So Eval in MCEVAL.scm

- So in the beginning of discussion, we had a simpler version on eval…
- Let's take a look at the bigger version…

Eval

```
(define (mc-eval exp env)
   (cond ((self-evaluating? exp) exp)
         ((variable? exp) (lookup-variable-value exp env))
         ((quoted? exp) (text-of-quotation exp))
         ((assignment? exp) (eval-assignment exp env))
         ((definition? exp) (eval-definition exp env))
         ((if? exp) (eval-if exp env))
         ((lambda? exp)
          (make-procedure (lambda-parameters exp)
                          (lambda-body exp)
                          env))
         ((begin? exp)
          (eval-sequence (begin-actions exp) env))
         ((cond? exp) (mc-eval (cond->if exp) env))
         ((application? exp)
          (mc-apply (mc-eval (operator exp) env)
                    (list-of-values (operands exp) env)))
         (else (error "Unknown expression type -- EVAL" exp))))
```

Interesting…

- Did you see what all the cond clauses have in common?

```
(cond (…
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-definition exp env))
        ((if? exp) (eval-if exp env))
        ((lambda? exp)
         (make-procedure (lambda-parameters exp)
                         (lambda-body exp)
                         env))
        ((begin? exp)
         (eval-sequence (begin-actions exp) env))
        ((cond? exp) (mc-eval (cond->if exp) env)) ...)
```

Special Forms…

- They're all special forms!
- Now you can create a special form!
- Like we said before, special forms don't follow the regular rules of evaluation, so they have their own clauses…

Apply

```
(define (mc-apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
         (eval-sequence
           (procedure-body procedure)
           (extend-environment
             (procedure-parameters procedure)
             arguments
             (procedure-environment procedure))))
        (else (error "Unknown procedure type -- APPLY" procedure))))
```

Cond Explicitly…

- Cy D. Fect doesn't like the way that cond clauses are evaluated in the MCE. He thinks its a waste of time to convert the cond statement into nested if statements before evaluating them. Cy would prefer that the evaluator directly handle the structure of the a cond statement.
- Your task is to define a function **eval-cond** that evaluates a cond expression within a given environment without making any new MCE if expressions.

```
;;inside eval's big cond statement ...
((cond? exp) (eval-cond exp env)) ...

(define (eval-cond exp env)
;;Your code goes here (helper functions may help...)
```

Cond…

```
(define (cond? exp)
    (tagged-list? exp 'cond))

(define (cond-clauses exp) (cdr exp))

(define (cond-else-clause? clause)
    (eq? (cond-predicate clause)
         'else))

(define (cond-predicate clause)
    (car clause))

(define (cond-actions clause)
    (cdr clause))

(define (cond->if exp)
    (expand-clauses (cond-clauses
    exp)))
```

```
(define (expand-clauses clauses)
    (if (null? clauses)
        'false                          ; no else clause
    (let ((first (car clauses))
          (rest (cdr clauses)))
      (if (cond-else-clause? first)
          (if (null? rest)
              (sequence->exp (cond-actions first))
              (error "ELSE clause isn't last
                     -- COND->IF“ clauses))
          (make-if (cond-predicate first)
                   (sequence->exp
                          (cond-actions first))
                   (expand-clauses rest))))))
```

Solution…

- ```
  (define (EVAL-COND exp ENV)
      (define (expand-clauses clauses)
              (if (null? clauses)
                  'false
                  (let ((first (car clauses))
                        (rest (cdr clauses)))
                    (if (cond-else-clause? first)
                        (EVAL (sequence->exp (cond-actions first)) ENV)
                        (IF (TRUE? (EVAL (cond-predicate first) ENV))
                            (EVAL (sequence->exp (cond-actions first))
                                  ENV)
                            (expand-clauses rest))))))
      (expand-clauses (cond-clauses exp)))
  ```

Lexical vs. Dynamic Scope

- One note on lexical vs dynamic scoping. Scoping refers to where we "point" our procedure calls. In lexical scoping, we point the frame to where *the procedure we call points to*, you should recognize this from Scheme. In dynamic scoping, you point your frame back to the last frame you were in. See the official lecture notes for implications.
  ```