

Meta Circular Evaluator (MCE)

General Features

A Scheme metacircular evaluator is an implementation of a Scheme interpreter developed using Scheme. Because we are writing Scheme using Scheme, our evaluator is necessarily metacircular in nature. This helps in some cases because we can easily use some of the primitive data types from the underlying Scheme interpreter.¹

There are three major components of a Scheme interpreter – the evaluator, the data, and the environment. All three of these topics have been covered by now, so they shouldn't be foreign to you². The evaluator follows the rules required to evaluate Scheme expressions properly. If you can evaluate Scheme expressions without an interpreter (by hand), you should have no problem understanding how the evaluator works. The data is important because they represent the objects the interpreter uses to perform operations and accomplish useful work. In the MCE, the only data type we must define ourselves is the procedure³, the others are borrowed from the underlying Scheme. The environment diagrams and their connection to normal evaluation is important to understanding how the MCE works because it must be represented in the MCE through code.

¹ If we had used another language we might have to design all of the data types from scratch.

² How well you have mastered these topics probably varies between individuals.

³ We need to build the environment and frames using a data abstraction, but I don't know if these fall under the category of "Scheme Data Types".

Evaluation

Determining How to Evaluate an Expression

When evaluating a Scheme expression by hand we first identify the type of expression and then act accordingly. The method used in SICP should be reminiscent of explicit dispatch with manifest types.⁴ By determining the type of expression, we can know how to evaluate it. This can be seen by examining the definition of `mc-eval`⁵:

```
(define (mc-eval exp env)
  (cond ((self-evaluating? exp) (eval-self-evaluating exp))
        ((variable? exp) (eval-variable exp env))
        ((quoted? exp) (eval-quoted exp))
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-definition exp env))
        ((if? exp) (eval-if exp env))
        ((lambda? exp) (eval-lambda exp env))
        ((begin? exp) (eval-begin exp env))
        ((cond? exp) (eval-cond exp env))
        ((application? exp) (eval-application exp env))
        (else (error "Unknown expression type -- EVAL" exp))))
```

Note that this doesn't look *exactly* like the code in SICP, but that's because my goal is to present the material from a different point of view. Initially you should notice the resemblance of this procedure with the explicit dispatch technique for manifest types. `Mc-eval` determines the type of expression and then applies the correct operation to evaluate the expression.⁶ A question for starters should be, "how can we determine the type of expression?" Let's define the predicates and find out how...

```
(define true #T)
(define false #F)

(define (mc-number? exp) (number? exp))
(define (mc-boolean? exp) (or (eq? exp true) (eq? exp false)))
(define (mc-string? exp) (string? exp))

(define (self-evaluating? exp)
  (cond ((mc-number? exp) #T) ; Numbers are self-evaluating
        ((mc-boolean? exp) #T) ; Boolean are self-evaluating
        ((mc-string? exp) #T) ; Strings are self-evaluating
        (else #F)))
```

⁴ This can be paralleled with manifest types and generic operators created to perform correctly given the data type. In this case, the evaluator is the operator and must correctly operate on the generic *expression* data type.

⁵ I've chosen to use "mc-" to distinguish the MCE's procedures from similarly named Scheme procedures. The book renames Scheme's procedures to avoid the conflict, but I don't find that method elegant. Also note that I change a few procedure names and bodies, and diverge from SICP's implementation in some places.

⁶ My document on *Representing Data Abstractions* covers manifest types and explicit dispatch.

The first three predicates – `mc-number?`, `mc-boolean?`, and `mc-string?` – would be useful if we had to define our own types. I've defined them to show that we don't necessarily have to be using Scheme's primitives, we just happen to be using them. In SICP they use two symbols `true` and `false` to let you know that the true and false values being used in the MCE are it's own interpretations of the values. This is a data abstraction, and to help it along I've defined `mc-boolean?`. Notice that I've defined something as self-evaluating if it's a number, boolean, or string. This is just like it was done in the document *Evaluating Scheme Expressions* when we were performing hand calculations!

```
(define (variable? exp)
  (symbol? exp)) ; A variable is just a symbol being evaluated.
```

To help with the remaining predicates, we can make a tool to help check the first element of an expression. SICP calls it `tagged-list?`, but I'd rather have a more expression related name, like `exp-tag?`.

```
(define (exp-tag? exp tag)
  (if (pair? exp)
      (eq? (car exp) tag)
      #F))
```

The rest of the predicates until `application?` are special forms and are indicated as follows:

```
(define (quoted? exp)
  (exp-tag? exp 'quote)) ; This starts with the symbol quote.

(define (assignment? exp)
  (exp-tag? exp 'set!)) ; This starts with the symbol set!.

(define (definition? exp)
  (exp-tag? exp 'define)) ; This starts with the symbol define.

(define (if? exp)
  (exp-tag? exp 'if)) ; This starts with the symbol if.

(define (lambda? exp)
  (exp-tag? exp 'lambda)) ; This starts with the symbol lambda.

(define (begin? exp)
  (exp-tag? exp 'begin)) ; This starts with the symbol begin.

(define (cond? exp)
  (exp-tag? exp 'cond)) ; This starts with the symbol cond.
```

Now we come to the last predicate which is *not* a special form. In fact, the only way we can tell if an expression is an application is by knowing that is a list which is not a special form! Putting it at the end tells us it isn't a special form, and now we just need to make sure it is a list:

```
(define (application? exp)
  (pair? exp)) ; Make sure this is a list...
```

Evaluating a Self-Evaluating Expression

Self-evaluating expression are the easiest to evaluate because they just evaluate to themselves:

```
(define (eval-self-evaluating exp) exp)
```

Evaluating a Variable Expression

Variables are more difficult to evaluate because they must be looked up in the environment to find the value to which they are bound. We have no problem with this however because we will be developing a data abstraction for environments and can expect `env-lookup-symbol` to be defined.

```
(define (eval-variable exp env) (env-lookup-symbol env exp))
```

Evaluating a Quoted Expression

Quoted expressions are quite simple to evaluate because they just delay the expression being quoted. The expression being delayed is the second element in the expression list.

```
(define (eval-quoted exp) (cadr exp))
```

Evaluating an Assignment

An assignment is similar to evaluating a variable because it must interact with the environment. In this case, we can assume the procedure `env-mutate-symbol!` will be defined to lookup a variable and change it's value in the environment. We must also remember that the expression must be evaluated before the assignment occurs.

```
(define (eval-assignment exp env)
  (let ((symbol (cadr exp))
        (value (mc-eval (caddr exp) env)))
    (env-mutate-symbol! env symbol value)
    'OK))
```

Evaluating a Definition

A definition either adds a new symbol to the local frame or uses an already existent symbol in the local frame and binds the given value to it. We'll assume the procedure `env-define-symbol!` does this for us. We must also remember that the procedure definition must be detected and handled properly. In that case we shall convert the procedure `define` into it's equivalent with the `lambda` expression. Again we must make sure the expression is evaluated before the symbol is bound.

```
(define (normal-define? exp)
  (symbol? (cadr exp)))

(define (definition-symbol exp)
  (if (normal-define?)
      (cadr exp) ; (define <symbol> <expression>)
      (caddr exp))) ; (define (<symbol> <p1> ... <pn>) <e1> ... <en>)
```

```
(define (make-lambda formal-params body)
  (cons 'lambda ; (lambda
    (cons formal-params ; (<symbol> <p1> ... <pn>)
      body))) ; <e1> ... <en>))

(define (definition-value exp)
  (if (normal-define? exp)
      (caddr exp)
      (make-lambda (caddr exp) ; Formal parameters
                   (caddr exp)))) ; Body (list of expressions)

(define (eval-definition exp env)
  (let ((symbol (definition-symbol exp))
        (value (mc-eval (definition-value exp) env)))
    (env-define-symbol! env symbol value)
    'OK))
```

Evaluating an If

Recall that the if has three parts – the predicate, the consequent, and the alternate. The alternate is optional, so we must make sure it works with or without one. The *if* special form works by first evaluating the predicate expression. If the predicate results in a true value, the consequent is evaluated, otherwise the alternate is evaluated if it exists. To insure that we treat anything that is *not false* as true, we define the two predicates *false?* and *true?*.

```
(define (false? value) (eq? false value))
(define (true? value) (not (false? value)))

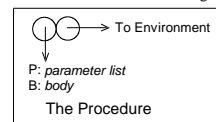
(define (if-predicate exp) (cadr exp))
(define (if-consequent exp) (caddr exp))
(define (if-alternate exp)
  (if (not (null? (caddr exp)))
      (caddr exp)
      #F))

(define (eval-if exp env)
  (let ((predicate (if-predicate exp))
        (consequent (if-consequent exp))
        (alternate (if-alternate exp)))
    (if (true? (mc-eval predicate env))
        (mc-eval consequent env)
        (if alternate (mc-eval alternate env))))))
```

Evaluating a Lambda

A lambda expression is special because it must create a procedure. Because procedures are closely tied to the environment and we are making our own environments (not using the ones in the underlying Scheme), we cannot use the underlying Scheme's procedures like we do the other primitive forms. When we create procedures from evaluating a lambda expression, we must

make our own type of procedure.⁷ Those procedures we don't create, namely primitive procedures, are still of the underlying Scheme's type. Let's define our procedure by make a data abstraction based on what we draw from our environment diagrams:



```
; The Constructor.
(define (make-procedure parameters body env)
  (list 'procedure parameters body env))

; The Predicate.
(define (compound-procedure? datum)
  (and (pair? procedure) (eq? (car procedure) 'procedure)))

; The Selectors.
(define (procedure-parameters procedure) (cadr procedure))
(define (procedure-body procedure) (caddr procedure))
(define (procedure-environment procedure) (caddr procedure))
```

Let's not forget to define a predicate for primitive Scheme procedures too:

```
(define (primitive-procedure? datum)
  (procedure? datum))
```

Now that we can create procedures, let's make the procedure that evaluates lambda expressions.

```
(define (eval-lambda exp env)
  (let ((parameters (cadr exp))
        (body (caddr exp)))
    (make-procedure parameters body env)))
```

⁷ This is only for creating procedures, we still use the primitive procedures defined in Scheme.

Evaluating a Begin

Begin expressions allow us to evaluate a list of expressions in sequence. The last evaluated expression's resultant value is the resultant value of the *begin* expression. We'll make a function called *expression-list-eval* to do this operation.⁸

```
(define (expression-list-eval exp-list env)
  (let ((exp (car exp-list))
        (rest (cdr exp-list)))
    (let ((value (mc-eval exp env)))
      (if (null? rest)
          value
          (expression-list-eval rest env)))))

(define (eval-begin exp env)
  (expression-list-eval (cdr exp) env))
```

Evaluating a Cond

The *cond* can be represented as a series of *if* expressions, which is how SICP does it. It could also be made separate of *if*, but then if we decided to change how conditional expressions were evaluated, we'd have to change both the *cond* and *if*, not just the *if*. To do this we simply convert the *cond* expression to an *if* expression and evaluate it as follows:

```
(define (make-if predicate consequent alternate)
  (list 'if predicate consequent alternate))

(define (make-begin exp-list)
  (cons 'begin exp-list))

(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)))
```

⁸ You should realize that it doesn't make any sense to have an empty list of expressions and that this would cause an error in normal Scheme as well.

```
(define (expand-clauses clauses)
  (if (null? clauses)
      false
      (let ((first (car clauses))
            (rest (cdr clauses)))
        (if (cond-else-clause? first)
            (if (null? rest)
                (make-begin (cond-actions first))
                (error "ELSE clause isn't last -- COND->IF"
                      clauses))
            (make-if (cond-predicate first)
                     (make-begin (cond-actions first))
                     (expand-clauses rest)))))

(define (eval-cond exp env)
  (mc-eval (cond->if exp) env))
```

Evaluating an Application

Now that we've taken care of all the special forms, we can work on applying procedures to actual parameters. First we evaluate all of the expressions in the list, then apply the procedure (operator) to the evaluated actual parameters (operands). If the operator turns out not to be a procedure, an error occurs. If there are a different number of formal parameters than actual parameters, an error also occurs.

To help us evaluate an application, we should make our own *mc-apply* procedure to imitate Scheme's *apply* procedure. *Mc-apply* should be capable of handling compound procedures and primitive procedures. At this point we should remember that when a procedure is called, a new frame is created with the formal parameters bound to the actual parameters, and this frame extends the environment pointed to by the procedure.⁹ For extending environments, we'll assume there is a procedure *env-extend*, which will be defined later. It takes two lists, one of the formal parameters and the other of actual parameters, and correctly extends the given environment.

```
(define (mc-apply procedure actual-params)
  (cond ((primitive-procedure? procedure)
        (apply procedure actual-params))
        (compound-procedure? procedure)
        (let ((exp-list (procedure-body procedure))
              (formal-params (procedure-parameters procedure))
              (let ((new-env (env-extend env
                                       formal-params
                                       actual-params))
                    (expression-list-eval exp-list new-env))))
          (else
           (error "Unknown procedure type -- APPLY" procedure))))))
```

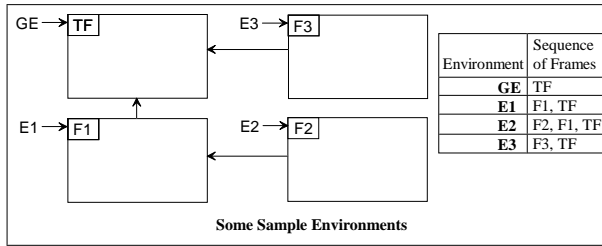
⁹ If this extended the same environment the procedure was called within it would be *dynamic scope* instead of *static scope*.

```
(define (eval-application exp env)
  (let ((evaluated-list (map (lambda (exp) (mc-eval exp env)) exp)))
    (let ((operator (car exp))
          (operands (cdr exp)))
      (mc-apply operator operands))))
```

The Environment & Frames

What Makes Up an Environment?

An environment, simply stated, is a bunch of frames organized in an order from most local to least local. This construct is designed to allow procedures to have local state when called. Some sample environments would be:



Making an Environment

An easy way of representing a sequence of environments would be to make a list of frames. The first frame in the list would be the most local frame, and the rest of the frames would be in order of increasing distance. Now let's make a data abstraction for an environment:

```
; The Constructor.
(define (make-environment frame old-environment)
  (cons frame old-environment))

; The Constant.
(define the-empty-environment '())

; The Predicate.
(define (empty-env? environment)
  (eq? the-empty-environment environment))

; The Selectors.
(define (enclosing-environment env) (cdr env))

(define (first-frame env) (car env))
```

Now's the time to define the procedures that manipulate the environment. We need to define *env-mutate-symbol!*, *env-define-symbol!*, and *env-extend*. These will in turn use procedures that will be defined later, namely *make-frame*, *frame-mutate-symbol!*, and *frame-add-binding!*.

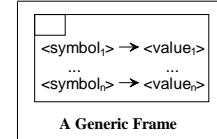
```
(define (env-mutate-symbol! env symbol value)
  (if (empty-env? env)
      (error "Unbound variable" symbol)
      (let ((frame (first-frame env))
            (rest (enclosing-environment env)))
        (if (frame-mutate-symbol! frame symbol value)
            #T
            (env-mutate-symbol! rest symbol value))))))

(define (env-define-symbol! env symbol value)
  (if (empty-env? env)
      (error "Empty environment" env)
      (let ((frame (first-frame env)))
        (if (not (frame-mutate-symbol! frame symbol value))
            (frame-add-binding! frame symbol value))))))

(define (env-extend env symbol-list value-list)
  (make-environment (make-frame symbol-list value-list)
                    env))
```

Making a Frame

Our definition of an environment wouldn't be complete without frames. What's a frame anyway? A frame is something that has a bunch of symbols bound to values:



This appears to be a table, so that's what we'll build as an abstract data type (except call it a frame). We don't have any constant frame values, so we can skip constants and predicates.

```
; The Constructor.
(define (make-frame symbols values)
  (if (= (length symbols) (length values))
      (cons symbols values)
      (error "Frame Construction Error" (cons symbols values))))

; The Selectors.
(define (frame-variables frame) (car frame))

(define (frame-values frame) (cdr frame))
```

```
; The Mutators.
(define (frame-add-binding! frame symbol value)
  (set-car! frame (cons symbol (car frame)))
  (set-cdr! frame (cons value (cdr frame))))

(define (frame-mutate-symbol! frame symbol value)
  (define (mutate-loop! symbols values symbol new-value)
    (cond ((null? symbols) #F)
          ((eq? (car symbols) symbol)
           (set-car! values new-value))
          (else
           (mutate-loop! (cdr symbols)
                         (cdr values)
                         symbol
                         value))))
  (mutate-loop! (frame-variables frame)
                (frame-values frame)
                symbol
                value))
```

The MCE Program¹⁰

Setting Up The Global Environment

Before we can use Scheme, we need to have an initial environment for adding frames, adding global variables, and finding primitive procedures and variables. We can do this by making an initial procedure that can be called to setup the global environment and bind it to a global variable.

```
(define primitive-procedure-names
  '(+ - / * cons car cdr eq? equal? list null?))

(define primitive-procedure-objects
  (list + - / * cons car cdr eq? equal? list null?))

(define (setup-environment)
  (let ((initial-env
        (env-extend the-empty-environment
                    primitive-procedure-names
                    primitive-procedure-objects)))
    initial-env))

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

Accepting Keyboard Input

Our interpreter wouldn't be the same if we couldn't type in input like we do in the normal Scheme interpreter. Would it? Of course not! That's why we need to make a procedure that makes a loop, reading in expressions from the keyboard, and then evaluating them with *mc-eval*. Every time we evaluate something at the top level of the loop, we'll be using the global environment, just like normal Scheme.

```
(define input-prompt ";; MC-Eval input:")
(define output-prompt ";; MC-Eval output:")

(define (prompt-for-input string)
  (newline) (newline) (display string) (newline))

(define (prompt-for-output string)
  (newline) (display string) (newline))

(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)))
```

Running The MCE

Now we could run the MCE as suggested in SICP by evaluating the following in sequence:

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

Or we could make a procedure to do this for us, and save some typing.¹¹ I'm assuming the variable *the-global-environment* is already bound to some value as it is in *eval.scm* in the CS61A *lib* directory.

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

This is extremely useful if you want to wipe the environment of the MCE without having to leave reload the *eval.scm* file into Scheme.

¹⁰I debated about the necessity of actually implementing the driver-loop part as well as the evaluator, but decided I had gone thus far and might as well finish the job.

¹¹This is what *eval.scm* does, so you only need to run the procedure *mce*.