CS 360: Programming Languages
Lecture 6: The Metacircular Evaluator

Geoffrey Mainland

Drexel University

Section 1

Overview of the Metacircular Evaluator

Metalinguistic abstraction

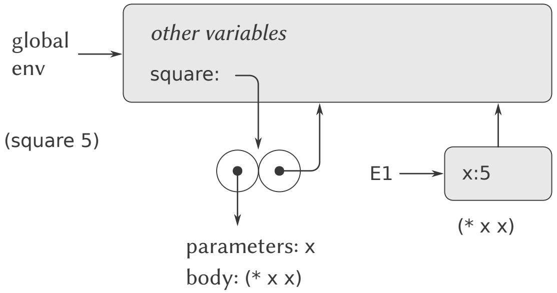
- SICP uses this term to refer to the process of establishing new languages.
- New problem domains often require new language abstractions. These new abstractions allow problems in a domain to be expressed more efficiently.
- Sometime this involves creating a new language; it is also possible to build new abstractions directly in a sufficiently expressive language.
- To get a high-level idea of how one might go about building a new language, we will write a Scheme interpreter. . . **in Racket**.
- This will also allow us to experiment with, e.g., new evaluation rules.

The metacircular evaluator

- Called “metacircular” because the evaluator for the language is written in the same language it evaluates.
- We will essentially formulate the environment model of evaluation from the last lecture as a Racket program.
- Having to make the evaluation model executable—that is, having to actually implement it as a program—forces us to really understand what is going on.

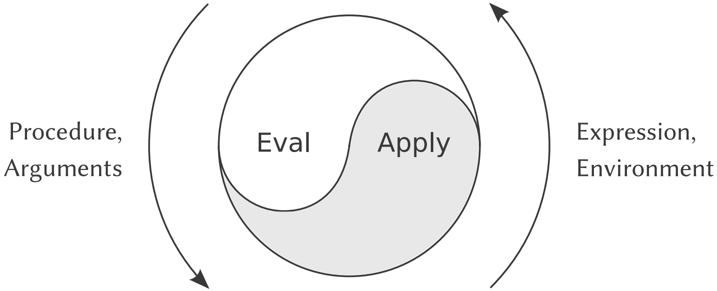
The basic rules of evaluation

1. To evaluate a compound expression (other than a special form), evaluate the subexpressions and then apply the value of the operator subexpression to the values of the operand subexpressions.
2. To apply a compound procedure to a set of arguments, evaluate the body of the procedure in a new environment. We construct this new environment by extending the environment part of the procedure object by a frame in which the procedure's formal parameters are bound to the procedure's arguments.



Implementing the basic rules of evaluation

- These two rules capture the essence of evaluation—the problem of evaluating an expression in an environment is reduced to the problem of applying procedures to arguments, which in turn is reduced to the problem of evaluating smaller expressions in new environments.
- We will capture these rules as Scheme functions `mceval` and `mcapply`.
- The "things" we are evaluating get smaller and smaller until we bottom out to. . . what?



The job of the evaluator.

If the evaluator already knows how to apply primitives, what is left to do?

- The evaluator must deal with compound expressions, such as `(+ 1 (* 2 3))`
- The evaluator must allow the use of variables.
- The evaluator must allow us to define *new* functions, i.e., compound procedures.
- The evaluator must also deal with special forms—they are exceptions to the rule for evaluating compound expressions.

The core of the evaluator: `mceval`

The function `mceval` takes as arguments... an expression to evaluate and an environment.

Each type of expression is evaluated by a different rule:

- **Primitive expressions**
  - **Self-evaluating expressions** (like numbers): return the expression itself.
  - **Variables**: look up the variable in the environment.
- **Special forms**
  - **Quoted expressions**: return the quoted expression.
  - **Assignment**: recursively call the evaluator to obtain the variable's new value, and then modify the environment.
  - **`if`**: evaluate the conditional, and then, depending on its value, evaluate one of the branches and return its value.
  - **`lambda`**: create a procedure object by packaging up the formal parameters and body of the lambda with the current evaluation environment.
  - **`begin`**: evaluate a sequence of expressions in order, returning the value of the last as the value of the special form.
  - **`cond`**: transform into nested `if` expressions and then evaluate that.

The core of the evaluator: `mceval`

The function `mceval` takes as arguments an expression to evaluate and an environment.

- **Combinations**
    - **A procedure application**: recursively evaluate the operator and the operands. The resulting procedure and arguments are passed to `mcapply`.
    - The function `mcapply` is responsible for actually "calling" the procedure.

Advice

- I will assume you have read SICP Chapters 4.1–4.2.
- It can take a while to wrap your head around what is going on in the metacircular evaluator. Always keep in mind the fact that there are **two language** which look similar—Racket, the language in which the evaluator is *implemented*, and "toy Scheme," the language *being evaluated* by the metacircular evaluator.
- Always be aware of *which interpreter* is performing evaluation—is it Racket or the interpreter *you* wrote?
- The code for the applicative version of the evaluator is in your Lab 2 repository.
- The code for the lazy version of the evaluator is in your Homework 2 repository.

Section 2

The Metacircular Evaluator

Plan for the next few lectures. . .

- We will first look at the code of the standard evaluator.
- We will then look at a *normal-order* variant.
- The code I will show you is slightly different from that found in the book—I have adapted it to Racket.
- First, let's look at Lab 2. . .
- **Lab 2 and Homework 2 are significantly more difficult than Homework 1 or Lab 1**. Many students find these two assignments to be the most difficult in the course.

The core of the evaluator: mceval

```
(define (mceval 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) (mceval (cond->if exp) env))
        ((application? exp)
         (mcapply (mceval (operator exp) env)
                  (list-of-values (operands exp) env)))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

mceval: Evaluating an application

```
(define (mceval exp env)
  (cond ...
        ((application? exp)
         (mcapply (mceval (operator exp) env)
                  (list-of-values (operands exp) env)))
        ...))
```

What do you think `list-of-values` does?

```
(define (list-of-values exps env)
  (if (no-operands? exps)
      '()
      (cons (mceval (first-operand exps) env)
            (list-of-values (rest-operands exps) env))))
(define (no-operands? ops) (null? ops))
(define (first-operand ops) (car ops))
(define (rest-operands ops) (cdr ops))
```

mceval: Evaluating an application

```
(define (list-of-values exps env)
  (if (no-operands? exps)
      '()
      (cons (mceval (first-operand exps) env)
            (list-of-values (rest-operands exps) env))))
```

Question: What is the order of evaluation in our metacircular interpreter?

mceval: Evaluating if

```
(define (mceval exp env)
  (cond ...
        ((if? exp) (eval-if exp env))
        ...))
```

How do you think the predicate `if?` is implemented?

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

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

How do you think the predicate `application?` is implemented?

mceval: Evaluating if

```
(define (eval-if exp env)
  (if (true? (mceval (if-predicate exp) env))
      (mceval (if-consequent exp) env)
      (mceval (if-alternative exp) env)))

(define (true? x) (not (eq? x false)))

(define (false? x) (eq? x false))
```

Question: What is the nature of truth? That is, how many things are true?

mceval: Evaluating if

```
(define (eval-if exp env)
  (if (true? (mceval (if-predicate exp) env))
      (mceval (if-consequent exp) env)
      (mceval (if-alternative exp) env)))

(define (if-predicate exp) (cadr exp))

(define (if-consequent exp) (caddr exp))

(define (if-alternative exp)
  (if (not (null? (cdddr exp)))
      (cadddr exp)
      'false))
```

Question: What is the value of an `if` expression where the predicate evaluates to `false` and there is no alternative?

mceval: Evaluating definitions and assignment

```
(define (mceval exp env)
  (cond ...
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-definition exp env))
        ...))
```

Question: How do you think `assignment?` and `definition?` are defined?

```
(define (assignment? exp)
  (tagged-list? exp 'set!))

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

mceval: Evaluating definitions and assignment

```scheme
(define (mceval exp env)
  (cond ...
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-definition exp env))
        ...))

(define (eval-assignment exp env)
  (set-variable-value! (assignment-variable exp)
                       (mceval (assignment-value exp) env)
                       env)
  'ok)

(define (eval-definition exp env)
  (define-variable! (definition-variable exp)
                    (mceval (definition-value exp) env)
                    env)
  'ok)
```

What do the `define` and `set!` special forms return? What does the language standard say they should return?

mceval: defining a variable with define

```
(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-mcar! vals val))
            (else (scan (cdr vars) (mcdr vals)))))
    (scan (frame-variables frame)
          (frame-values frame))))
```

How do you think environments and frames are represented?
How do you think the implementation of `set!` differs from the implementation of `define`?

Brief interlude: `mcons` and friends

- Racket has *mutable* cons cells, which are created with `mcons` and modified with `set-mcar!` and `set-mcdr!`.
- R5RS Scheme has `set-car!` and `set-cdr!`.
- Why do you think Racket changed this?

mceval: setting a variable with set!

```
(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-mcar! vals val))
            (else (scan (cdr vars) (mcdr vals)))))
    (if (eq? env the-empty-environment)
        (error "Unbound variable -- SET!" var)
        (let ((frame (first-frame env)))
          (scan (frame-variables frame)
                (frame-values frame)))))
  (env-loop env))
```

mceval: defining a variable with define (recap)

```
(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-mcar! vals val))
            (else (scan (cdr vars) (mcdr vals)))))
    (scan (frame-variables frame)
          (frame-values frame))))
```

mceval: evaluating variables

```
(define (mceval exp env)
  (cond ...
        ((variable? exp) (lookup-variable-value exp env))
        ...)
(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))
             (mcar vals))
            (else (scan (cdr vars) (mcdr vals)))))
    (if (eq? env the-empty-environment)
        (error "Unbound variable" var)
        (let ((frame (first-frame env)))
          (scan (frame-variables frame)
                (frame-values frame)))))
  (env-loop env))
```

mceval: setting a variable with set! (recap)

```
(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-mcar! vals val))
            (else (scan (cdr vars) (mcdr vals)))))
    (if (eq? env the-empty-environment)
        (error "Unbound variable -- SET!" var)
        (let ((frame (first-frame env)))
          (scan (frame-variables frame)
                (frame-values frame)))))
  (env-loop env))
```

mceval: evaluating quote

```
(define (mceval exp env)
  (cond ...
        ((quoted? exp) (text-of-quotation exp))
        ...)

(define (text-of-quotation exp) (cadr exp))
```

mceval: Why so much indirection?

Definitions have the form

```
(define <var> <value>)
```

or

```
(define (<var> <parameter1> ... <parametern>)
  <body>)
```

The second form corresponds to

```
(define <var>
  (lambda (<parameter1> ... <parametern>)
  <body>))
```

We need to handle both forms.

`mceval`: Why so much indirection?

Note that a `lambda`'s body is a *list* of expressions!

```
(define (eval-definition exp env)
  (define-variable! (definition-variable exp)
                    (mceval (definition-value exp) env)
                    env)
  'ok)

(define (definition-variable exp)
  (if (symbol? (cadr exp))
      (cadr exp)
      (caadr exp)))

(define (definition-value exp)
  (if (symbol? (cadr exp))
      (caddr exp)
      (make-lambda (cdadr exp)
                   (cddr exp))))

(define (make-lambda parameters body)
  (cons 'lambda (cons parameters body)))
```

mceval: evaluating lambda

```
(define (mceval exp env)
  (cond ...
        ((lambda? exp)
         (make-procedure (lambda-parameters exp)
                         (lambda-body exp)
                         env))
        ...))

(define (make-procedure parameters body env)
  (list 'procedure parameters body env))
```

mceval: evaluating begin

```
(define (mceval exp env)
  (cond ...
        ((begin? exp)
         (eval-sequence (begin-actions exp) env))
        ...))

(define (eval-sequence exps env)
  (cond ((last-exp? exps) (mceval (first-exp exps) env))
        (else (mceval (first-exp exps) env)
              (eval-sequence (rest-exps exps) env))))
```

For applications, we used `list-of-values` to evaluate operands. What was the order of evaluation? What is the order of evaluation for begin? How is it enforced?

mceval: evaluating applications

What do the evaluation rules say about how to evaluate compound expressions?

```
(define (mceval exp env)
  (cond ...
        ((application? exp)
         (mcapply (mceval (operator exp) env)
                  (list-of-values (operands exp) env)))
        ...))
```

The core of the evaluator: mcapply

The function mcapply takes as arguments. . . a procedure and a list of arguments to which the procedure must be applied.

```
(define (mcapply 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))))
```

The code for apply-primitive-procedure is slightly tricky...

Applying primitives

```
(define (apply-primitive-procedure proc args)
  (apply (primitive-implementation proc) args))
```

Recommended: figure out what the representation of primitives is in the interpreter.

One more special form to deal with...

In our high-level description of the evaluator, we said that we would evaluate cond by translating it into a nested sequence of `if` expressions.
For example, we can translate this

```
(cond ((> x 0) x)
      ((= x 0) (display 'zero) 0)
      (else (- x)))
```

into the equivalent expression

```
(if (> x 0)
    x
    (if (= x 0)
        (begin (display 'zero)
               0)
        (- x)))
```

Expanding cond

- A cond expression consist of the symbol cond and a list of *clauses*.
- A clause consists of a predicate and actions. The actions are a *list* of expressions—there is an implicit begin in the actions portion of a clause.

```
(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)))
```

Expanding cond: expand-clauses

- The real work is done by expand-clauses.
- sequence->exp converts a sequence of expressions to a single expression. Why do we need it?

```
(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))))))

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

Expanding cond: converting a sequence of expressions to an expression

```
(define (last-exp? seq) (null? (cdr seq)))
(define (first-exp seq) (car seq))
(define (rest-exps seq) (cdr seq))

(define (make-begin seq) (cons 'begin seq))

(define (sequence->exp seq)
  (cond ((null? seq) seq)
        ((last-exp? seq) (first-exp seq))
        (else (make-begin seq))))
```

What about let?

How would you add `let` to the interpreter? Could you implement it in a similar fashion to cond, i.e., by rewriting a `let` expression into a form that the interpreter already knows how to handle?

We want to convert this:

```
(let ((<var1> <exp1>) ... (<varn> <expn>))
  <body>)
```

into this:

```
((lambda (<var1> ... <varn>)
   <body>)
 <exp1>
 ...
 <expn>)
```

You will implement a different approach in Lab 2.