METACIRCULAR EVALUATOR 17

GEORGE WANG
gswang.cs61a@gmail.com
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley

July 20, 2010

Introduction

One thing that comes up is that you might think that it's best to write an evaluator that is designed to read in a easy language, but also to use a really powerful language that is capable of dealing with complex structures and dealing with symbols. However, it turns out that these two languages can be the same language. Furthermore, we can use Scheme for both of these, since its structure of syntax is simple to parse, and it is great at handling symbols. One important question to address is why it's useful to write a Scheme compiler in Scheme, since obviously you'd have to have something that reads that in the first place.

First of all, there's a good pedagogical reason for doing it. By doing so, you will have a much stronger image of how something like that works, as well as have a good idea of how Scheme itself works. We'll be implementing the page of code that is essentially our rules for evaluating environment diagrams.

Secondly, it turns out much of Scheme itself is implemented in Scheme. Once you get out of memory management and such, it turns out a lot of STk is actually written in Scheme as well.

Thirdly, we will be able to add new features and change fundamentally the way Scheme works for us.

Lastly, there's a very very powerful idea, which is called UNIVERSALITY. Universality means we can write one program that's equivalent to all other programs that can ever be written. At the hardware level, this is the idea that made general-purpose computers possible. It used to be that they built a separate machine, from scratch, for every new problem. An intermediate stage was a machine that had a patchboard so you could rewire it, effectively changing it into a different machine for each problem, without having to re-manufacture it. The final step was a single machine that accepted a program as data so that it can do any problem without rewiring. This is the ultimate in data-directed programming.

1 First Version: Echo

Let's write this first version, that doesn't do anything interesting, except read your input and spit it back out at you. You also can't actually quit out of it...

```
;cs61a/lectures/mceval/mc-eval-step1.scm
(define (input-loop)
  (display "=61A=> ")
  (print (mc-eval (read)))
  (input-loop)

(define (mc-eval exp) exp)
```

So this is pretty dumb, but it introduces two things. It has a way of reading input from the user, as well as a procedure that is supposed to evaluate what an expression evaluates into.

2 Second Version: Self-Evaluating, Primitives

Let's look at the things that Scheme can handle. We know that there are kind of atoms (self-evaluating and symbols) as well as lists (special forms and procedure calls). In this second version, let's just try to build a basic calculator. In other words, in terms of self-evaluating things, we'll say that for numbers, their value and their expression are the same. We also say that we translate primitives that we see into primitives in Scheme.

```
;cs61a/lectures/mceval/mc-eval-step2.scm
(define (input-loop)
  (display "=61A=> ")
  (let ((input (read)))
    (if (equal? input 'exit)
        (print "Au Revoir!")
        (begin (print (mc-eval input))
               (input-loop)))))

(define (mc-eval exp)
   (cond ((number? exp) exp)
         ((primitive? exp) (eval exp)) ;Calling STk!
         ((list? exp) (mc-apply (mc-eval (car exp))
                                (map mc-eval (cdr exp))))
         (else (error "UNKNOWN expression"))))

;;Call a procedure!
(define (mc-apply fn args)
  (do-magic fn args))

;;Make STk do it!
(define (do-magic fn args)
  (apply fn args))

(define (primitive? exp)
  (or (eq? exp '+)   (eq? exp '-)   (eq? exp '/)    (eq? exp '*)
      (eq? exp 'car) (eq? exp 'cdr) (eq? exp 'cons) (eq? exp 'null?)
      ;; more))
```

2.1 booleans and if

Assume the definitions that aren't explicitly stated from now on have not changed. In our version of Scheme, we'll be using #t and #f to represent the true and false values.

```
;cs61a/lectures/mceval/mc-eval-step2.5.scm
(define (input-loop)
  (display "=61A=> ")
  (let ((input (read)))
    (if (equal? input 'exit)
        (print "Au Revoir!")
        (begin (print (mc-eval input))
               (input-loop)))))

(define (mc-eval exp)
   (cond ((self-evaluating? exp) exp)
         ((primitive? exp) (eval exp)) ;Calling STk!
         ((if-exp? exp)
           (if (not (eq? (mc-eval (cadr exp)) #f)) ;Everything true except for false
               (mc-eval (caddr exp))
               (mc-eval (cadddr exp))))
         ((list? exp) (mc-apply (mc-eval (car exp))
                                (map mc-eval (cdr exp))))
        (else (error "UNKNOWN expression"))))

(define (self-evaluating? exp)
  (or (number? exp)
      (boolean? exp)))

(define (boolean? exp)
  (or (eq? exp #t)
      (eq? exp #f)))

(define (if-exp? exp)
  (and (list? exp)
       (eq? (car exp) 'if)))
```

3 Global Variables, Define

Now we're going to start introducing the concept of environments. In the previous version, we had a bunch of primitives explictly defined. There was no real concept of an environment, or any concept of frames. In this version, we will do a very basic global frame, as well as give define. Note, this is not the substitution or the environment model, we still do not have the ability to create procedures.

```
;cs61a/lectures/mceval/mc-eval-step3.scm
(define (mc-eval exp)
  (cond
    ((self-evaluating? exp) exp)
    ((variable? exp) (lookup-variable-value exp))
    ((if-exp? exp)
      (if (not (eq? (mc-eval (cadr exp)) #f))
          (mc-eval (caddr exp))
          (mc-eval (cadddr exp))))
    ((definition? exp)
      (define-variable! (cadr exp) ;;Variable
                        (mc-eval (caddr exp)))) ;Value
    ((list? exp) (mc-apply (mc-eval (car exp))
                           (map mc-eval (cdr exp))))
    (else (error "UNKNOWN expression"))))
```

```
(define (definition? exp)
  (eq? (car exp) 'define))

(define (variable? exp)
   (symbol? exp))
```

So truth be told, the way that the book does frames is kind of stupid. The smart way they could have done it is to just make it an association list of variables and values. Instead, a FRAME is a pair of parallel lists. In other words, the 4th variable name in the first list corresponds to the 4th value in the second list. Thus, in the two procedures below to lookup variable names and define variable names, it looks through both of the lists at once.

```
(define the-global
  (cons (list '+ '- '/ '* 'car 'cdr 'cons 'null? 'nil 'yell 'square 'factorial)
        (list +  -  /  *  car  cdr  cons  null?  nil  yell  square  factorial)))

(define (define-variable! var val)
  (define (scan vars vals)
    (cond
      ((null? vars)         ; Create a new binding
       (set-car! the-global (cons var (car the-global)))
       (set-cdr! the-global (cons val (cdr the-global))))
      ((eq? var (car vars)) ;Change a binding
       (set-car! vals val))
      (else                 ; Keep Looking
       (scan (cdr vars) (cdr vals)))))
  (scan (car the-global)
        (cdr the-global))
  var)

(define (lookup-variable-value var)
  (define (scan vars vals)
    (cond ((null? vars) (error "Unbound Variable"))
          ((eq? var (car vars)) (car vals))
          (else (scan (cdr vars) (cdr vals)))))
  (scan (car the-global)
        (cdr the-global)))
```

3.1 Adding begin

We're going to add begin. But the point of this isn't to do begin, per se, but to be able to evaluate a sequence of expressions. This will let us move towards making procedures in the next step. Let's look at how that's done:

```
;cs61a/lectures/mceval/mc-eval-step3.5.scm
(define (mc-eval exp)
  (cond
    ((self-evaluating? exp) exp)
    ((variable? exp) (lookup-variable-value exp))
    ((if-exp? exp)
      (if (not (eq? (mc-eval (cadr exp)) #f))
          (mc-eval (caddr exp))
          (mc-eval (cadddr exp))))
    ((begin-exp? exp)
      (eval-sequence (cdr exp)))
    ((definition? exp)
      (define-variable! (cadr exp) ;;Variable
                        (mc-eval (caddr exp)))) ;Value
    ((list? exp) (mc-apply (mc-eval (car exp)) (map mc-eval (cdr exp))))
```

```
    (else (error "UNKNOWN expression"))))

(define (begin-exp? exp)
  (eq? (car exp) 'begin))

(define (eval-sequence exps)
  (cond ((null? (cdr exps)) (mc-eval (car exps)))
    (else
      (mc-eval (car exps))
      (eval-sequence (cdr exps)))))
```

4 Adding Procedures and Environments

Okay, now we're going to add the concept of environments and lambda expressions. As a reminder, let's think about environment diagrams again. With procedures in an Environment Diagram, you know that they have three properties that they know about. They have the parameters, the body, and the frame that it's associated with. Be careful, this introduces the idea of a circular list. The frames contain a procedure, which in turn can contain a frame. If you aren't careful about this, you could run into an infinite loop if you try to actually view it, since there's no way for STk to display them.

The important thing here to note is that now, expressions can no longer be evaluated in a void. Expressions must have an associated environment. Before we look at the full version in all its gory detail, let's look at a simplified version:

```
;;;;; In file cs61a/lectures/4.1/micro.scm
(define (scheme)
  (display "> ")
  (print (mceval (read) the-global-environment))
  (scheme))

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

(define (mcapply proc args)
  (if (primitive? proc)
      (do-magic proc args)
      (mceval (body proc) (extend-environment (formals proc) ;variables
                                              args           ;values
                                              (proc-env proc) ))))
```

Note the awesome simplicity of this! In about 15 lines of code, we have all the the big ideas a evaluator for Scheme, written in Scheme. Yes, it's true that it has tons of helper functions that it's calling, but all of them are self-documenting. We know basically everything about what we need to know in order to implement the (almost) final version. This is missing only a few special forms as well as the ability to quote.

```
; cs61a/lectures/mceval/mc-eval-step2.5.scm
(define (input-loop)
  (display "=61A=> ")
  (let ((input (read)))
    (if (equal? input 'exit)
        (print "Au Revoir!")
        (begin (print (mc-eval input))
               (input-loop)))))

(define (mc-eval exp env)
  (cond ((self-evaluating? exp) exp)
  ((variable? exp) (lookup-variable-value exp env))
  ((if-exp? exp)
    (if (not (eq? (mc-eval (cadr exp) env) #f))
        (mc-eval (caddr exp) env)
        (mc-eval (cadddr exp) env)))
  ((begin-exp? exp)
   (eval-sequence (cdr exp) env))
  ((definition? exp)
    (define-variable! (cadr exp)
                      (mc-eval (caddr exp) env)
                      env))
  ((lambda-exp? exp)
    (make-procedure (cadr exp) ;formals
                    (cddr exp) ;body
                    env))
  ((list? exp) (mc-apply (mc-eval (car exp) env)
                         (map (lambda (arg-exp) (mc-eval arg-exp env))
                              (cdr exp))))
  (else (error "UNKNOWN expression"))))

(define (mc-apply fn args)
  (cond ((lambda-proc? fn)
          (eval-sequence (body fn)
                         (extend-environment (params fn)
                                             args
                                             (env fn))))
        (else (do-magic fn args))))
;;;;;;;;;;;;;;;;;;;;;;;;; Procedure ADT ;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define (make-procedure params body env) (list 'procedure params body env))
(define (params p)(cadr p))
(define (body p)  (caddr p))
(define (env p)   (cadddr p))
(define (lambda-proc? p) (and (list? p) (eq? (car p) 'procedure)))
;;;;;;;;;;;;;;;;;;;;;;;;; Helper Procedures ;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define (set-exp? exp)    (eq? (car exp) 'set!))
(define (lambda-exp? exp) (eq? (car exp) 'lambda))
(define (begin-exp? exp)  (eq? (car exp) 'begin))
(define (if-exp? exp)          (and (list? exp)   (eq? (car exp) 'if)))
(define (boolean? exp)         (or (eq? exp #t) (eq? exp #f)))
(define (do-magic fn args)     (apply fn args))
(define (definition? exp)      (eq? (car exp) 'define))
(define (variable? exp)        (symbol? exp))
(define (self-evaluating? exp) (or (number? exp)  (boolean? exp)))
(define (eval-sequence exps env)
  (cond ((null? (cdr exps))
         (mc-eval (car exps) env))
        (else (mc-eval (car exps) env)
              (eval-sequence (cdr exps) env))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;; Environment Related ;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define (extend-environment vars vals base-env)
  (cons (cons vars vals) base-env))
(define (define-variable! var val env)
  (define first-frame (car env))
  (define (scan vars vals)
    (cond ((null? vars)
     (set-car! first-frame (cons var (car first-frame)))
     (set-cdr! first-frame (cons val (cdr first-frame))))
    ((eq? var (car vars))
     (set-car! vals val))
    (else
     (scan (cdr vars) (cdr vals)))))
  (scan (car first-frame) (cdr first-frame))
  var)

(define the-global-frame
  (cons (list '+ '- '/ '* 'car 'cdr 'cons 'nil 'list)
        (list + - / * car cdr cons nil list)))
(define the-global-env (cons the-global-frame nil))
(define (lookup-variable-value var env)
  (define first-frame (car env))
  (define (scan vars vals)
    (cond ((null? vars)
             (if (eq? env the-global-env)
                 (error "Unbound Variable")
                 (lookup-variable-value var (cdr env))))
          ((eq? var (car vars)) (car vals))
          (else (scan (cdr vars) (cdr vals)))))
  (scan (car first-frame)
        (cdr first-frame)))
```