CS61A Lecture 24

2011-08-01

Colleen Lewis

Goals

- Increase comfort with the meta-circular evaluator (MCE)
- Identify inefficiency
- See efficiency improvement using `analyze`
- Connect the ideas in `analyze` to compiling

`mce` review

- You could define new functions in `mce`?
  - A. True
  - B. False

How many calls to `mc-eval`?

```
;;; M-Eval input:
(define (fact n)
  (if (= n 0)
      1
      (* n (fact (- n 1)))))
```

A) 1 B) 2 C) 3 D) 4 E) 5

Does this make any calls to `mc-apply`?

A) Yes B) NO!!!

How many calls to `mc-eval`?

```
;;; M-Eval input:
(define (simple x) x)
```

A)1 B) 2 C) 3 D) 4 E) 5

```
(define (mc-eval exp env)
  (display (list 'mc-eval 'exp: exp))
  (newline)
  (cond ((self-evaluating? exp) exp)
```

How many calls to `mc-eval`?

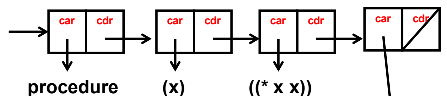
```
;;; M-Eval input:

(simple 5)
```

A)1 B) 2 C) 3 D) 4 E) 5

REVIEW What is a procedure?

```
STk> (mc-eval '(lambda (x) (* x x)) '(((a) 3)))
(procedure (x) ((* x x)) (((a) 3)))
```



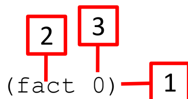
Global
a: 3

car cdr

Params: x
Body: (+ x x)

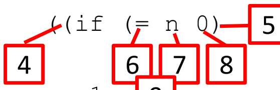
((a) . (3))

How many calls to mc-eval? DEMO




```
(fact 0)
(procedure
   (n)
   (if (= n 0)
        1
     (* n (fact (- n 1)))))
 env)
```

1 4 5



9

calls to mc-eval

```
(mc-eval exp: (fact 0))
(mc-eval exp: fact)
(mc-eval exp: 0)
(mc-eval exp:
      (if (= n 0) 1 (* n (fact (- n 1)))))
(mc-eval exp: (= n 0))
(mc-eval exp: =)
(mc-eval exp: 0)
(mc-eval exp: n)
(mc-eval exp: 1)
```

(fact 1)

```
(fact 1)
(mc-eval exp: (fact 1))
(mc-eval exp: fact)
(mc-eval exp: 1)

(mc-eval exp: (if (= n 0) 1 (* n (fact (- n 1)))))
(mc-eval exp: (= n 0))
(mc-eval exp: =)
(mc-eval exp: 0)
(mc-eval exp: n)
(mc-eval exp: (* n (fact (- n 1))))
(mc-eval exp: *)
(mc-eval exp: (fact (- n 1)))
(mc-eval exp: fact)
(mc-eval exp: (- n 1))
(mc-eval exp: -)
(mc-eval exp: 1)
(mc-eval exp: n)
```

```
(mc-eval exp:
(if (= n 0) 1 (* n
     (fact (- n 1)))))
(mc-eval exp: (= n 0))
(mc-eval exp: =)
(mc-eval exp: 0)
(mc-eval exp: n)
(mc-eval exp: 1)
(mc-eval exp: n)
```

(fact 1)

```
(fact 1)
(mc-eval exp: (fact 1))
(mc-eval exp: fact)
(mc-eval exp: 1)

(mc-eval exp: (if (= n 0) 1 (* n (fact (- n 1)))))
(mc-eval exp: (= n 0))
(mc-eval exp: =)
(mc-eval exp: 0)
(mc-eval exp: n)
(mc-eval exp: (* n (fact (- n 1))))
(mc-eval exp: *)
(mc-eval exp: (fact (- n 1)))
(mc-eval exp: fact)
(mc-eval exp: (- n 1))
(mc-eval exp: -)
(mc-eval exp: 1)
(mc-eval exp: n)
```

```
(mc-eval exp:
(if (= n 0) 1 (* n
     (fact (- n 1)))))
(mc-eval exp: (= n 0))
(mc-eval exp: =)
(mc-eval exp: 0)
(mc-eval exp: n)
(mc-eval exp: 1)
(mc-eval exp: n)
```

(fact 2)

```
(mc-eval exp: (fact 2))
(mc-eval exp: fact)
(mc-eval exp: 2)
(mc-eval exp: (if (= n 0) 1 (* n (fact (- n 1)))))
(mc-eval exp: (= n 0))
(mc-eval exp: =)
(mc-eval exp: 0)
(mc-eval exp: n)
(mc-eval exp: (* n (fact (- n 1))))
(mc-eval exp: *)
(mc-eval exp: (fact (- n 1)))
(mc-eval exp: fact)
(mc-eval exp: (- n 1))
(mc-eval exp: -)
(mc-eval exp: 1)
(mc-eval exp: n)
```

```
(mc-eval exp: (if (= n 0) 1 (* n (fact (- n
1)))))
(mc-eval exp: (= n 0))
(mc-eval exp: =)
(mc-eval exp: 0)
(mc-eval exp: n)
(mc-eval exp: (* n (fact (- n 1))))
(mc-eval exp: *)
(mc-eval exp: (fact (- n 1)))
(mc-eval exp: fact)
(mc-eval exp: (- n 1))
(mc-eval exp: -)
(mc-eval exp: 1)
(mc-eval exp: n)
```

```
(mc-eval exp: (if (= n 0) 1
        (* n (fact (- n 1)))))
(mc-eval exp: (= n 0))
(mc-eval exp: =)
(mc-eval exp: 0)
(mc-eval exp: n)
(mc-eval exp: 1)
(mc-eval exp: n)
(mc-eval exp: n)
```

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

Each call to `mc-eval` could have a lot of sub-calls!

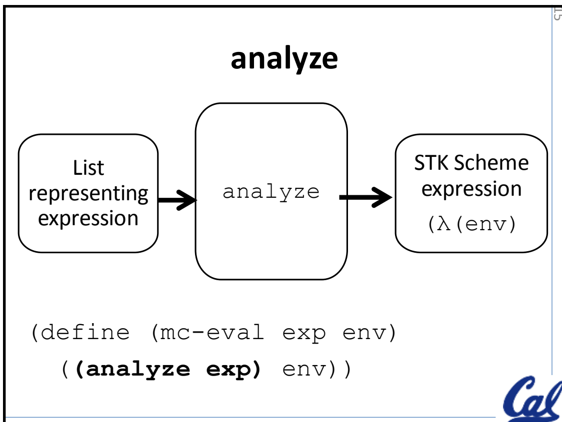
Most didn't depend upon the environment so I could do in advance

analzye

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

What is the domain and range of `analyze`?

A. Domain: function   Range: function
B. Domain: expression   Range: function
C. Domain: function   Range: expression
D. Domain: expression   Range: expression
E. Other




```
(define (analyze exp)
  (cond
    ((self-evaluating? exp)
    ((quoted? exp) …
    ((variable? exp) …
    ((assignment? exp) …
    ((definition? exp) …
    ((if? exp) …
    ((lambda? exp) …
    ((begin? exp) …
    ((cond? exp) …
    ((application? exp) …
    (else  (error "Unknown" exp))))
```

analyze

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

```
(define (mc-eval exp env)
  (cond
    ((self-evaluating? exp) exp)
    ((quoted? exp) (text-of-quotation exp))
    ((variable? exp) (lookup-variable-value exp env))
    …
(define (analyze exp)
  (cond
    ((self-evaluating? exp)
              (analyze-self-evaluating exp))
    ((quoted? exp)
              (analyze-quoted exp))
    ((variable? exp)
              (analyze-variable exp))
    …
```

```
(define (mc-eval exp env)
  (cond
    ((self-evaluating? exp) exp) ...
(define (analyze exp)
  (cond
    ((self-evaluating? exp)
              (analyze-self-evaluating exp))...

(define (analyze-self-evaluating exp)
  (lambda (env) exp))
```

Is the domain and range correct?   A) Yes   B) No

List representing expression → `analyze` → STK Scheme expression (λ(env)

```
(define (mc-eval exp env)
  (cond ...
     ((variable? exp) (lookup-variable-value exp env))
     ...
(define (analyze exp)
  (cond ...
     ((variable? exp) (analyze-variable exp))
     ...
(define (analyze-variable exp)
     ;; write this!
  (lambda (env) (lookup-variable-value exp env)))
```

Is the thing you returned entirely scheme (it only needs to be interpreted by STk)? A) Yes B) No C) ???

```
(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))
          (car vals))
       (else (scan (cdr vars) (cdr 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))
```

analyze-quoted

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

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

(define (analyze exp)
  (cond ...
     ((quoted? exp)
               (analyze-quoted exp))
     ...
```

Two versions of analyze-quoted

```
(define (analyze-quoted_v1 exp)
  (lambda (env) (text-of-quotation exp)))

(define (analyze-quoted_v2 exp)
  (let ((qval (text-of-quotation exp)))
    (lambda (env) qval)))
```

A) Only v1 works
B) Only v2 works
C) v1 is better
D) v2 is better
E) They are the same

Write analyze-if

```
(define (mc-eval exp env)
  (cond ...
     ((if? exp) (eval-if exp env))
     ...
(define (analyze exp)
  (cond ...
     ((if? exp) (analyze-if exp))  ...
     ...
(define (eval-if exp env)
  (if (true? (mc-eval (if-predicate exp) env))
      (mc-eval (if-consequent exp) env)
      (mc-eval (if-alternative exp) env)))
```

Is the analyzed code faster if it is run multiple times? A)Y B)N

Write analyze-if

(analyze '(if #t 3 4))

```
(define (analyze exp)
  (cond ((self-evaluating? exp)
                (analyze-self-evaluating exp))
        ((if? exp) (analyze-if exp))...

(define (analyze-if exp)
  (let ((pproc (analyze (if-predicate exp)))
        (cproc (analyze (if-consequent exp)))
        (aproc (analyze (if-alternative exp))))
    (lambda (env)
      (if (true? (pproc env))
          (cproc env)
          (aproc env)))))
```

Do we save time using the analyzing mce?

`(sent-sum '(1 2 3 4 5 6 7 8))`

A) Yes B) No C)??

`(sent-sum '())`

A) Yes B) No C)??

`(list (+ 2 3) (+ 4 5) (+ 2 3))`

A) Yes B) No C)??

`(list (sq 2) (sq 3) (sq 4))`

A) Yes B) No C)??

Compiling Java

```
cory [344] ~/javaexample # emacs HelloWorld.java &
cory [346] ~/javaexample # javac HelloWorld.java
cory [347] ~/javaexample # java HelloWorld
hello world
cory [348] ~/javaexample # ls
HelloWorld.java        HelloWorld.class
```

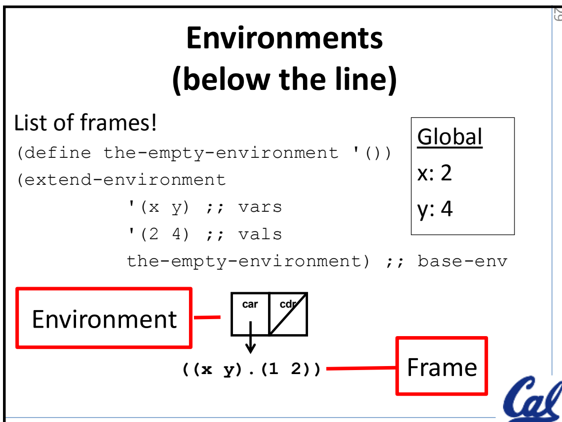
Compile

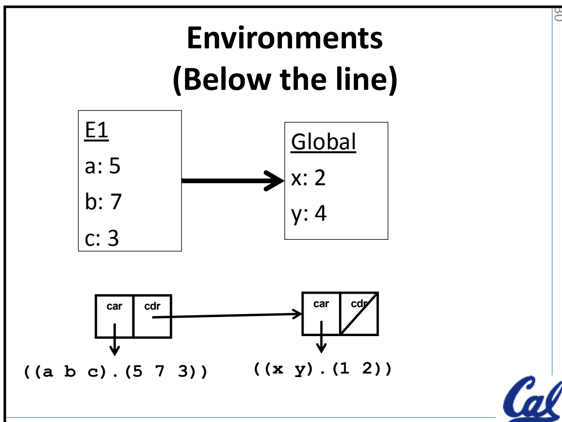
Run

Makes an executable file

Compilers

- Analyze syntax
- Make something that can be run on a computer
- Provide optimization
- Provide useful feedback to the programmer when there are errors





Solutions

How many calls to `mc-eval`?

```
;;; M-Eval input:
(define (simple x) x)
```

A)1 B) 2 C) 3 D) 4 E) 5

```
(define (mc-eval exp env)
  (display (list 'mc-eval 'exp: exp))
  (newline)
  (cond ((self-evaluating? exp) exp)

(mc-eval exp: (define (simple x) x))
(mc-eval exp: (lambda (x) x))
```

How many calls to `mc-eval`?

```
;;; M-Eval input:
(simple 5)
```

A)1 B) 2 C) 3 D) 4 E) 5

```
(mc-eval exp: (simple 5))
(mc-eval exp: simple)
(mc-eval exp: 5)
(mc-eval exp: x)
```

```
(define (mc-eval exp env)
  (cond ...
     ((variable? exp) (lookup-variable-value exp env))
     ...
(define (analyze exp)
  (cond ...
     ((variable? exp) (analyze-variable exp))
     ...

(define (analyze-variable exp)
     ;; write this!
  (lambda (env) (lookup-variable-value exp env)))
```

Is the thing you returned entirely scheme (it only needs to be interpreted by STk)? A) Yes B) No C) ???

`analyze-if` solution

```
(define (analyze-if exp)
  (let ((pproc (analyze (if-predicate exp)))
        (cproc (analyze (if-consequent exp)))
        (aproc (analyze (if-alternative exp))))
    (lambda (env)
      (if (true? (pproc env))
          (cproc env)
          (aproc env)))))
```

Is the thing you returned entirely scheme (it only needs to be interpreted by STk)? A) Yes B) No C) ???

`(analyze '(if #t 3 4))`

```
(define (analyze exp)
  (cond ((self-evaluating? exp)
               (analyze-self-evaluating exp))
        ((if? exp) (analyze-if exp))...

(define (analyze-if exp)
  (let ((pproc (analyze (if-predicate exp)))
        (cproc (analyze (if-consequent exp)))
        (aproc (analyze (if-alternative exp))))
    (lambda (env)
      (if (true? ( (λ(e) #t) env))
          ( (λ(e) 3) env)
          ( (λ(e) 4) env)))))
```