Meta-Evaluation
MIT 6.001 Recitation

Interpretation

Look at the code below and be amazed that it is so small.

Can you identify the 4 steps of the application?

Where is the double-bubble code?

How is the look-up rule implemented?

How do you see that subexpressions are evaluated first?

Warm up

Trace the execution of the following scheme expressions in the interpreter. Draw the box-and-arrow diagram of the environment model.

(+ 1 1)

(define x (+ 7 8))

(+ x 2)

(define double (lamda(x)(* x 2)))

(double x)

Factorial

Trace the execution of

```
(define fact
  (lambda(n)
    (if (= n 1)
      1
      (* n (fact (- n 1))))))

(fact 3)
```

Extensions

How would you implement the following features:

```
let
cond
set!
```

A counter that traces how many times a compound procedure is called

Make the evaluation of subexpressions truly random!

Note that we can also change the *semantics* of our language. For example, we could decide that all the subexpressions of the if expression are evaluated.

How would you make define return the value?

Interpreter Code

```
(define (eval exp env)
  (cond ((number? exp) exp)
      ((symbol? exp) (lookup exp env))
      ((define? exp) (eval-define exp env))
      ((if? exp) (eval-if exp env))
      ((lambda? exp) (eval-lambda exp env))
      ((application? exp) (apply (eval (car exp) env)
                                 (map (lambda (e) (eval e env)) (cdr exp))))
      (else (error "unknown expression " exp))))

 (define (apply operator operands)
  (cond ((primitive? operator) (scheme-apply (get-scheme-procedure operator) operands))
        ((compound? operator)
          (eval (body operator)
            (extend-env-with-new-frame (parameters operator) operands (env operator))))
        (else (error "operator not a procedure: " operator))))

(define (eval-if exp env)
  (let ((predicate (cadr exp)) (consequent (caddr exp)) (alternative (cadddr exp)))
    (let ((test (eval predicate env)))
      (cond ((eq? test #t) (eval consequent env))
            ((eq? test #f) (eval alternative env))
            (else (error ”predicate not boolean: ")

(define (eval-lambda exp env)
  (let ((args (cadr exp)) (body (caddr exp)))
    (make-compound args body env)))

;; ADT that implements the “double bubble”
(define compound-tag 'compound)
(define (make-compound parameters body env) (list compound-tag parameters body env))
(define (compound? exp) (tag-check exp compound-tag))
(define (parameters compound) (cadr compound))
(define (body compound) (caddr compound))
(define (env compound) (cadddr compound))

(define (application? e) (pair? e))

; Environment = list<table>

(define (extend-env-with-new-frame names values env)
  (let ((new-frame (make-table)))
    (make-bindings! names values new-frame)
    (cons new-frame env)))

(define (make-bindings! names values table)
  (for-each
    (lambda (name value) (table-put! table name value)) names values))
; the initial global environment

(define GE
  (extend-env-with-new-frame (list 'plus* 'greater*)
                             (list (make-primitive +) (make-primitive >)) nil))
; lookup searches the list of frames for the first match
(define (lookup name env)
  (if (null? env) (error "unbound variable: " name)
    (let ((binding (table-get (car env) name)))
      (if (null? binding) (lookup name (cdr env)) (binding-value binding)))))

; define changes the first frame in the environment
(define (eval-define exp env)
  (let ((name (cadr exp))   (defined-to-be (caddr exp)))
    (table-put! (car env) name (eval defined-to-be env))
    'undefined))
```