1. Arithmetic calculator

(define (tag-check e sym) (and (pair? e) (eq? (car e) sym)))
(define (sum? e) (tag-check e 'plus*))

(define (eval exp)
    (cond
        ((number? exp) exp)
        ((sum? exp) (eval-sum exp))
        (else
            (error "unknown expression " exp))))

(define (eval-sum exp)
    (+ (eval (cadr exp)) (eval (caddr exp))))

(eval '(plus* 24 (plus* 5 6)))
2. Names

(define (define? exp) (tag-check exp 'define*))

(define (eval exp)
  (cond
    ((number? exp) exp)
    ((sum? exp) (eval-sum exp))
    ((symbol? exp) (lookup exp))
    ((define? exp) (eval-define exp))
    (else
      (error "unknown expression " exp)))))

; variation on table ADT from March 2 lecture (only difference is
; that table-get returns a binding, while original version
; returned a value):
; make-table    void -> table
; table-get    table, symbol -> (binding | null)
; table-put!   table, symbol, anytype -> undef
; binding-value binding -> anytype

(define environment (make-table))

(define (lookup name)
  (let ((binding (table-get environment name)))
    (if (null? binding)
      (error "unbound variable: " name)
      (binding-value binding)))))

(define (eval-define exp)
  (let ((name (cadr exp))
        (defined-to-be (caddr exp)))
    (table-put! environment name (eval defined-to-be))
    'undefined))

(eval '(define* x* (plus* 4 5)))
(eval '(* x* 2))
3. Conditionals and if

(define (greater? exp) (tag-check exp 'greater*))
(define (if? exp) (tag-check exp 'if*))

(define (eval exp)
  (cond
   ((number? exp) exp)
   ((sum? exp) (eval-sum exp))
   ((symbol? exp) (lookup exp))
   ((define? exp) (eval-define exp))
   ((greater? exp) (eval-greater exp))
   ((if? exp) (eval-if exp))
   (else
    (error "unknown expression " exp))))

(define (eval-greater exp)
  (> (eval (cadr exp)) (eval (caddr exp)))))

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

(eval '(define* y* 9))
(eval '(if* (greater* y* 6) (plus* y* 2) 15))

; Index to procedures that have not changed:
; procedure        page  line
;   sum?            1   4
;   eval-sum       1   13
;   lookup         2   22
;   define?        2   3
;   eval-define    2   28
4. Store operators in the environment

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

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

;; rename scheme's apply so we can reuse the name
(define scheme-apply apply)

(define (apply operator operands)
  (if (primitive? operator)
      (scheme-apply (get-scheme-procedure operator) operands)
      (error "operator not a procedure: " operator)))

;; primitive: an ADT that stores scheme procedures

(define prim-tag 'primitive)
(define (make-primitive scheme-proc) (list prim-tag scheme-proc))
(define (primitive? e) (tag-check e prim-tag))
(define (get-scheme-procedure prim) (cadr prim))

(define environment (make-table))
(table-put! environment 'plus* (make-primitive +))
(table-put! environment 'greater* (make-primitive >))
(table-put! environment 'true* #t)

(eval '(define* z* 9))
(eval '(plus* 9 6))
(eval '(if* true* 10 15))

; Index to procedures that have not changed:
; procedure evaluator line
;  lookup 2 22
;  define? 2 3
;  eval-define 2 28
;  if? 3 4
;  eval-if 3 20
5. Environment as explicit parameter

;This change is boring! Exactly the same functionality as #4.

(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))
   ((application? exp) (apply (eval (car exp) env)
                              (map (lambda (e) (eval e env))
                                   (cdr exp))))
   (else
    (error "unknown expression " exp))))

(define (lookup name env)
  (let ((binding (table-get env name)))
    (if (null? binding)
        (error "unbound variable: " name)
        (binding-value binding))))

(define (eval-define exp env)
  (let ((name (cadr exp))
        (defined-to-be (caddr exp)))
    (table-put! env name (eval defined-to-be env))
    'undefined))

(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 "val not boolean: " predicate))))))

(eval '(define* z* (plus* 4 5)) environment)
(eval '(if* (greater* z* 6) 10 15) environment)

Index to procedures that have not changed:

<table>
<thead>
<tr>
<th>procedure</th>
<th>evaluator</th>
<th>line</th>
</tr>
</thead>
<tbody>
<tr>
<td>define?</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>if?</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>application?</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>apply</td>
<td>4</td>
<td>19</td>
</tr>
</tbody>
</table>
6. Defining new procedures

(define (lambda? e) (tag-check e 'lambda*))

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

(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)))))

;; 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))
; Environment model code (part of eval 6)

; 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))

(eval '(define* twice* (lambda* (x*) (plus* x* x*)))) GE
(eval '(twice* 4) GE)

Index to procedures that have not changed:
  procedure   evaluator  line
  define?     2           3
  if?         3           4
  application? 4           3
  eval-i