```
;;; A Scheme Interpreter

;; eval takes an expression and an environment, checks the type of the
;; expression and evaluates it appropriately.  In your C++/Java
;; implementation, you would implement each branch of the cond as a
;; virtual function in the appropriate subclass of class Node or
;; in the appropriate subclass of class Special.
(define (eval exp env)
  (cond ((symbol? exp) (lookup exp env))                ; variable
        ((not (pair? exp)) exp)                         ; constant
        ((eq? (car exp) 'closure) exp)                  ; closure
        ((eq? (car exp) 'quote) (cadr exp))             ; quote
        ((eq? (car exp) 'lambda)                        ; lambda
         (list 'closure exp env))
        ((eq? (car exp) 'begin)                         ; begin
         (evalbody (cdr exp) env))
        ((eq? (car exp) 'if)                            ; if
         (if (eval (cadr exp) env)
             (eval (caddr exp) env)
             (eval (cadddr exp) env)))
        ((eq? (car exp) 'let) (eval-let exp env))       ; let
        ((eq? (car exp) 'cond) (eval-cond exp env))     ; cond
        ((eq? (car exp) 'define) (eval-define exp env)) ; define
        ((eq? (car exp) 'set!) (eval-set! exp env))     ; set!
        (else (let ((call (evallist exp env)))          ; apply
                (apply (car call) (cdr call))))
        ))

;; Make sure you understand how the lookup function traverses the
;; environment.
;; The best way to implement lookup is to define a class Environement
;; and to make lookup() a method of that class.
(define (lookup exp env)
  (if (null? env)
      ; If we didn't find it in env, look in the scheme48 environment.
      ; The interpreter writtin in C++/Java must implement builtin-eval.
      (builtin-eval exp (interaction-environment))
      (let ((binding (assq exp (car env))))
        (if (pair? binding)
            (cadr binding)
            (lookup exp (cdr env))))))

;; Evaluate a list of expressions and return the list of results.
(define (evallist l env)
  (map (lambda (x) (eval x env)) l))

;; Evaluate a list of expressions and return the result of the last one.
;; This is inefficient but it works.
(define (evalbody l env)
  (car (reverse (evallist l env))))

;; Evaluate the expressions in the let-bindings, put a new scope in
;; front of env and use evalbody to evaluate the body.
;; Evaluating the expressions in the bindings can be done quite
;; elegantly using map.
(define (eval-let exp env)
  (let ((let-env (cons (map (lambda (x) (list (car x) (eval (cadr x) env)))
                            (cadr exp))
                       env)))
    (evalbody (cddr exp) let-env)))

;; eval-cond is implemented here as a single function, but it might be
;; easier to use a local helper function for evaluating a single case.
(define (eval-cond exp env)
  (cond ((null? exp) '())
        ((eq? (car exp) 'cond) (eval-cond (cdr exp) env))
        (else (let ((test (caar exp))
                    (body (cdar exp)))
                (if (eq? test 'else)
                    (if (null? body) '() (evalbody body env))
                    (let ((val (eval test env)))
                      (if val
```

```
                        (if (null? body) val (evalbody body env))
                        (eval-cond (cdr exp) env))))))))

;; Construct a lambda expression for function definitions and call
;; eval recursively to evaluate the RHS of the definition.
(define (eval-define exp env)
  ;; The local helper function def! adds a binding to the environment
  ;; as explained on the handout.
  ;; The best way to implement def! is as a method of a class Environment.
  (define (def! name value)
    (let ((binding (assq name (car env))))
      (if (pair? binding)
          (set-cdr! binding (list value))
          (set-car! env (cons (list name value) (car env))))))

  (if (symbol? (cadr exp))
      (def! (cadr exp) (eval (caddr exp) env))
      (let ((name (caadr exp))
            (parm (cdadr exp))
            (body (cddr exp)))
        (let ((value (eval (cons 'lambda (cons parm body)) env)))
          (def! name value)))))

;; Find the definition of the variable if it exists and assign the value.
(define (eval-set! exp env)
  ;; The local helper function lookup is similar to lookup for assigning
  ;; to a variable.  Alternatively, one could modify lookup to return the
  ;; binding, use it here, and change the case for symbols in eval.
  ;; The best way to implement lookup is as a method of a class Environment.
  (define (lookup name env)
    (if (null? env)
        '()
        (let ((binding (assq name (car env))))
          (if (pair? binding)
              binding
              (lookup name (cdr env))))))

  (let ((binding (lookup (cadr exp) env)))
    (if (pair? binding)
        (set-cdr! binding (list (eval (caddr exp) env)))
        binding)))

;; The first argument of apply is either a scheme48 closure, in which
;; case the built-in function procedure? returns #t, or a closure of
;; the form (closure (lambda ...) env).
;; The easiest way to implement apply is as a virtual function in the
;; Node and Special hierarchy, which reports an error for all classes
;; other than Regular.
(define (apply f args)
  ;; A helper function for pairing up parameters and arguments.
  ;; Instead of proper error handling if the lists are not the same length,
  ;; extra arguments in l2 are ignored, missing arguments default to '().
  (define (pairup l1 l2)
    (cond ((null? l1) '())
          ((null? l2) (cons (list (car l1) '()) (pairup (cdr l1) '())))
          (else (cons (list (car l1) (car l2)) (pairup (cdr l1) (cdr l2))))))

  ; Call builtin-apply from top-level.scm for built-in functions.
  (cond ((procedure? f) (builtin-apply f args))
        ; If it's not a closure this is an error.  No proper error handling.
        ((not (and (pair? f) (eq? (car f) 'closure))) '())
        ; Now we got a closure, take it apart and call evalbody.
        (else (let ((fun (cadr f))
                    (env (caddr f)))
                (let ((parm (cadr fun))
                      (body (cddr fun)))
                  (evalbody body (cons (pairup parm args) env)))))))
```