Pseudo Code

Here's the pseudo-code for the Scheme interpreter presented in a functional style. In an object-oriented style, each one of the cases in the if-then-else chain is a method of one of the classes from the parse tree node hierarchy.

```
(define (eval exp env)

    ; non-recursive cases

        if 'exp' is a symbol,
            look it up in 'env' and return its value

        if 'exp' is a constant (not a symbol and not a pair),
            return it

        if 'exp' is of the form '(quote e)',
            return e

        if 'exp' is of the form '(lambda ...)',
            return the closure new Closure(exp, env)

    ; recursive cases that don't modify the environment

        if 'exp' is of the form '(begin e1 ... en)',
             recursively call eval on e1 ... en and
             return the result of evaluating en

        if 'exp' is of the form '(if b t e)',
             evaluate b
             if it is not #f, evaluate t and return the result
             if it is #f, evaluate e and return the result

        if 'exp' is of the form '(cond (b1 ...) ... (bn ...)),
             evaluate b1 to bn until we find a condition bi that's not #f,
             if the i'th case of the cond expression is of the form '(bi)',
                 simply return bi
             if the i'th case of the cond expression is '(bi e1 ... en)'
                 recursively evaluate e1 ... en and
                 return the result of evaluating en
```

```
; recursive cases that modify the environment

    if 'exp' is of the form '(define x e)',
         evaluate e, let's call the result e1
         lookup x in the current scope
         if a binding for x exists already, store e1 as the new value,
         if no binding for x exists,
             add the pair (x e1) as the first element of the first
             association lists into the environment

    if 'exp' is of the form '(define (x p1 ... pn) b1 ... bm)',
         construct the lambda expression
             (lambda (p1 ... pn) b1 ... bm)
         proceed as for the definition
             (define x (lambda (p1 ... pn) b1 ... bm))

    if 'exp' is of the form '(set! x e)',
         evaluate e, let's call the result e1
         lookup x in the environment (not just in the current scope)
         if a binding for x exists already, store e1 as the new value,
         if no binding for x exists,
             this is an ERROR and
             the behavior of set! is undefined

    if 'exp' is of the form '(set-car! x e)',
         evaluate x, let's call the result x1
         evaluate e, let's call the result e1
         if x1 is not a Cons node, this is an ERROR
         store e1 in the car field of x1

    if 'exp' is of the form '(set-cdr! x e)',
         evaluate x, let's call the result x1
         evaluate e, let's call the result e1
         if x1 is not a Cons node, this is an ERROR
         store e1 in the cdr field of x1
```

```
    ; recursive cases that construct a new environment

        if 'exp' is of the form '(let ((x1 e1) ... (xn en)) b1 ... bm)',
            recursively evaluate e1 ... en, let's call the results
            e1' ... en'
            construct an association list ((x1 e1') ... (xn en'))
            this is the scope of the let body
            create a new environment 'env1' by 'cons'-ing this association
                list in front of 'env'
            recursively evaluate b1 ... bm in the new environment env1
            return the result of evaluating bm

        otherwise, i.e., if 'exp' is of the form '(f a1 ... an)',
            ; we handled all the special cases before, so this must
            ; be a function call now
            recursively evaluate f, a1, ... an
            let's call the results f', a1', ... an'
            call apply with
                f' as first argument and
                the list (a1' ... an') as second argument

(define (apply cl args)

        ; cl must be a closure, i.e., either an object
        ; of class Closure or an object of class BuiltIn
        if cl is an object of class BuiltIn
            implement the function in terms of the corresponding
            C++ (or Java) operation
        if cl is a Closure
            extract the lambda expression l and the environment e
            the lambda expression is of the form
                '(lambda (p1 ... pn) b1 ... bm)
            extract the parameters p1 ... pn
            build an association list ((p1 a1) ... (pn an)),
                where a1 ... an are the 'args'
            this is the scope for the lambda expression
            construct a new environment by 'cons'-ing this scope in
            front of 'e' (the environment found in the closure)
            recursively evaluate b1 ... bm in this new environment
            return the result of evaluating bm
```

That's all. In fact, the Scheme interpreter written in Scheme isn't any longer than this pseudo-code.