The Metacircular Evaluator

Guowei Lv

September 25, 2020

Contents

1 The Core of the Evaluator **1**
1.1 Eval 1
1.2 Apply 3
1.3 Procedure arguments 4
1.4 Conditionals 4
1.5 Sequences 4
1.6 Assignments and definitions 5

2 Representing Expressions **5**
2.1 The only self-evaluating items are numbers and strings: 6
2.2 Variables are represented by symbols: 6
2.3 Quotations have the form `(quote <text-of-quotation>)`: . 6
2.4 Assignments have the form `(set! <var> <value>)`: 6
2.5 Definitions have the form 6
2.6 Lambda expressions are lists that begin with the symbol `lambda`: 7
2.7 Conditionals begin with `if` and have a predicate, a consequent, and an (optional) alternative. If the expression has no alternative part, we provide `false` as the alternative. 7
2.8 `Begin` packages a sequence of expressions into a single expression. 8
2.9 A procedure application is any compound expression that is not one of the above expression types. The `car` of the expression is the operator, and the `cdr` is the list of operands: . . . 9
2.10 Derived expressions 9

3 Evaluator Data Structures **11**
3.1 Testing of predicates 11
3.2 Representing procedures 11
3.3 Operations on Environments 12

4 Running the Evaluator as a Program **15**

5 source code **17**

1 The Core of the Evaluator

The evaluation process can be described as the interplay between two procedures: `eval` and `apply`.

1.1 Eval

`Eval` takes as arguments an expression and an environment. It classifies the expression and directs its evaluation. `Eval` is structured as a case analysis of the syntactic type of the expression to be evaluated. In order to keep the procedure general, we express the determination of the type of an expression abstractly, making no commitment to any particular representation for the various types of expressions. Each type of expression has a predicate that tests for it and an abstract means for selecting its parts. This *abstract syntax* makes it easy to see how we can change the syntax of the language by using the same evaluator, but with a different collection of syntax procedures.

- *Primitive Expressions*
  - For self-evaluating expressions, such as numbers, `eval` returns the expression itself.
  - `Eval` must look up variables in the environment to find their values.
- *Special Forms*
  - For quoted expressions, `eval` returns the expression that was quoted.
  - An assignment to (or a definition of) a variable must recursively call `eval` to compute the new value to be associated with the variable. The environment must be modified to change (or create) the binding of the variable.

  - An `if` expression requires special processing of its parts, so as to evaluate the consequent if the predicate is true, and otherwise to evaluate the alternative.
  - A `lambda` expression must be transformed into an applicable procedure by packing together the parameters and body specified by the `lambda` expression with the environment of the evaluation.
  - A `begin` expression requires evaluating its sequence of expressions in the order in which they appear.
  - A case analysis (`cond`) is transformed into a nest of if expressions and then evaluated.

- *Combinations*

  - For a procedure application, `eval` must recursively evaluate the operator part and the operands of the combination. The resulting procedure and arguments are passed to `apply`, which handles the actual procedure application.

Here is the definition of `eval`:

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotation exp))
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-definition exp env))
        ((if? exp) (eval-if exp env))
        ((lambda? exp)
         (make-procedure (lambda-parameters exp)
                         (lambda-body exp)
                         env))
        ((begin? exp)
         (eval-sequence (begin-actions exp) env))
        ((cond? exp) (eval (cond->if exp) env))
        ((application? exp)
         (apply (eval (operator exp) env)
                (list-of-values (operands exp) env)))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

For clarity, `eval` has been implemented as a case analysis using `cond`. The disadvantage of this is that our procedure handles only a few distinguishable types of expressions, and no new ones can be defined without editing the definition of `eval`. In most Lisp implementations, dispatching on the type of an expression is done in a data-directed style. This allows a user to add new types of expressions that `eval` can distinguish, without modifying the definition of `eval` itself.

1.2 Apply

`Apply` takes two arguments, a procedure and a list of arguments to which the procedure should be applied. `Apply` classifies procedures into two kinds: It calls `apply-primitive-procedure` to apply primitives; it applies compound procedures by sequentially evaluating the expressions that make up the body of the procedure. The environment for the evaluation of the body of a compound procedure is constructed by extending the base environment carried by the procedure to include a frame that binds the parameters of the procedure to the arguments to which the procedure is to be applied. Here is the definition of `apply`:

```
(define (apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
         (eval-sequence
          (procedure-body procedure)
          (extend-environment
           (procedure-parameters procedure)
           arguments
           (procedure-environment procedure))))
        (else
         (error
          "Unknown procedure type -- APPLY" procedure))))
```

1.3 Procedure arguments

When `eval` processes a procedure application, it uses `list-of-values` to produce the list of arguments to which the procedure is to be applied. `List-of-values` takes as an argument the operands of the combination. It evaluates each operand and returns a list of the corresponding values:

```
(define (list-of-values exps env)
  (if (no-operands? exps)
      '()
      (cons (eval (first-operand exps) env)
            (list-of-values (rest-operands exps) env))))
```

1.4 Conditionals

`Eval-if` evaluates the predicate part of an `if` expression in the given environment. If the result is true, `eval-if` evaluates the consequent, otherwise it evaluates the alternative:

```
(define (eval-if exp env)
  (if (true? (eval (if-predicate exp) env))
      (eval (if-consequent exp) env)
      (eval (if-alternative exp) env)))
```

The use of `true?` in `eval-if` highlights the issue of the connection between an implemented language and an implementation language. The `if-predicate` is evaluated in the language being implemented and thus yields a value in that language. The interpreter predicate `true?` translates that value into a value that can be tested by the `if` in the implementation language: The metacircular representation of truth might not be the same oas that of the underlying Scheme.

1.5 Sequences

`Eval-sequence` is used by `apply` to evaluate the sequence of expressions in a procedure body and by `eval` to evaluate the sequence of expressions in a `begin` expression. It takes as arguments a sequence of expressions and an environment, and evaluates the expressions in the order in which they occur. The value returned is the value of the final expression.

```
(define (eval-sequence exps env)
  (cond ((last-exp? exps) (eval (first-exp exps) env))
        (else (eval (first-exp exps) env)
              (eval-sequence (rest-exps exps) env))))
```

1.6 Assignments and definitions

The following procedure handles assignments to variables. It calls `eval` to find the value to be assigned and transmits the variable and the resulting value to `set-variable-value!` to be installed in the designated environment.

```
(define (eval-assignment exp env)
  (set-variable-value! (assignment-variable exp)
                       (eval (assignment-value exp) env)
                       env)
  'ok)
```

Definitions of variables are handled in a similar manner.

```
(define (eval-definition exp env)
  (define-variable! (definition-variable exp)
                    (eval (definition-value exp) env)
                    env)
  'ok)
```

We have chosen here to return the symbol `ok` as the value of an assignment or a definition.

2 Representing Expressions

The evaluator is reminiscent of the symbolic differentiation program discussed in section 2.3.2. both programs operate on symbolic expressions. In both programs, the result of operating on a compound expression is determined by operating recursively on the pieces of the expression and combining the results in a way that depends on the type of the expression. In both programs we used data abstraction to decouple the general rules of operation from the details of how expressions are represented. In the differentiation program this meant that the same differentiation procedure could deal with algebraic expressions in prefix form, in infix form, or in some other form. For the evaluator, this means that the syntax of the language being evaluated is determined solely by the procedures that classify and extract pieces of expressions.

Here is the specification of the syntax of our language:

2.1 The only self-evaluating items are numbers and strings:

```
(define (self-evaluating? exp)
  (cond ((number? exp) true)
        ((string? exp) true)
        (else false)))
```

2.2 Variables are represented by symbols:

```
(define (variable? exp) (symbol? exp))
```

2.3 Quotations have the form `(quote <text-of-quotation>)`:

```
(define (quoted? exp)
  (tagged-list? exp 'quote))

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

`Quoted?` is defined in terms of the procedure `tagged-list?`, which identifies lists beginning with a designated symbol:

```
(define (tagged-list? exp tag)
  (if (pair? exp)
      (eq? (car exp) tag)
      false))
```

2.4 Assignments have the form `(set! <var> <value>)`:

```
(define (assignment? exp)
  (tagged-list? exp 'set!))

(define (assignment-variable exp) (cadr exp))

(define (assignment-value exp) (caddr exp))
```

2.5 Definitions have the form

`(define <var> <value>)`
or the form
`(define (<var> <param1> <param2> ... <paramn>) <body>)`
The latter form (standard procedure definition) is syntactic sugar for

```
(define <var>
  (lambda (<param1> ... <paramn>)
   <body>))
```

The corresponding syntax procedures are the following:

```
(define (definition? exp) (tagged-list? exp 'define))

(define (definition-variable exp)
  (if (symbol? (cadr exp))
      (cadr exp)
      (caadr exp)))

(define (definition-value exp)
  (if (symbol? (cadr exp))
      (caddr exp)
      (make-lambda (cdadr exp)   ;formal parameters
                   (cddr exp)))) ;body
```

2.6 Lambda expressions are lists that begin with the symbol `lambda`:

```
(define (lambda? exp) (tagged-list? exp 'lambda))

(define (lambda-parameters exp) (cadr exp))

(define (lambda-body exp) (cddr exp))

(define (make-lambda parameters body)
  (cons 'lambda (cons parameters body)))
```

2.7 Conditionals begin with `if` and have a predicate, a consequent, and an (optional) alternative. If the expression has no alternative part, we provide `false` as the alternative.

```
(define (if? exp) (tagged-list? exp 'if))

(define (if-predicate exp) (cadr exp))

(define (if-consequent exp) (caddr exp))

(define (if-alternative exp)
  (if (not (null? (cdddr exp)))
      (cadddr exp)
      'false))
```

```
(define (make-if predicate consequent alternative)
  (list 'if predicate consequent alternative))
```

2.8 Begin packages a sequence of expressions into a single expression.

We include syntax operations on `begin` expressions to extract the actual sequence from the `begin` expression, as well as selectors that return the first expression and the rest of the expressions in the sequence.

```
(define (begin? exp) (tagged-list? exp 'begin))

(define (begin-actions exp) (cdr exp))

(define (last-exp? seq) (null? (cdr seq)))

(define (first-exp seq) (car seq))

(define (rest-exps seq) (cdr seq))

(define (make-begin seq) (cons 'begin seq))

(define (sequence->exp seq)
  (cond ((null? seq) seq)
        ((last-exp? seq) (first-exp seq))
        (else (make-begin seq))))
```

We also include a constructor `sequence->exp` (for use by `cond-if`) that transforms a sequence into a single expression, using `begin` if necessary:

```
(define (sequence->exp seq)
  (cond ((null? seq) seq)
        ((last-exp? seq) (first-exp seq))
        (else (make-begin seq))))

(define (make-begin seq) (cons 'begin seq))
```

2.9 A procedure application is any compound expression that is not one of the above expression types. The car of the expression is the operator, and the cdr is the list of operands:

```
(define (application? exp) (pair? exp))

(define (operator exp) (car exp))

(define (operands exp) (cdr exp))

(define (no-operands? ops) (null? ops))

(define (first-operand ops) (car ops))

(define (rest-operands ops) (cdr ops))
```

2.10 Derived expressions

Some special forms in our languages can be defined in terms of expressions involving other special forms, rather than being implemented directly. One example is `cond`, which can be implemented as a nest of `if` expressions. For example, we can reduce the problem of evaluating the expression

```
(cond ((> x 0) x)
      ((= x 0) (display 'zero) 0)
      (else (- x)))
```

to the problem of evaluating the following expression involving `if` and `begin` expressions:

```
(if (> x 0)
    x
    (if (= x 0)
        (begin (display 'zero)
               0)
        (- x)))
```

Implementing the evaluation of `cond` in this way simplifies the evaluator because it reduces the number of special forms for which the evaluation process must be explicitly specified.

We include syntax procedures that extract the parts of a `cond` expression and procedure `cond->if` that transforms `cond` expressions into `if` expressions. A case analysis begins with `cond` and has a list of predicate-action clauses. A clause is an `else` clause if its predicate is the symbol `else`.

```
(define (cond? exp) (tagged-list? exp 'cond))

(define (cond-clauses exp) (cdr exp))

(define (cond-predicate clause) (car clause))

(define (cond-actions clause) (cdr clause))

(define (cond-else-clause? clause)
  (eq? (cond-predicate clause) 'else))

(define (cond->if exp)
  (expland-clauses (cond-clauses exp)))

(define (expand-clauses clauses)
  (if (null? clauses)
      'false
      (let ((first (car clauses))
            (rest (cdr clauses)))
        (if (cond-else-clause? first)
            (if (null? rest)
                (sequence->exp (cond-actions first))
                (error "ELSE clause isn't last -- COND->IF"
                       clauses))
            (make-if (cond-predicate first)
                     (sequence->exp (cond-actions first))
                     (expand-clauses rest))))))
```

Expressions (such as `cond`) that we choose to implement as syntactic transformations are called *derived expressions*. `Let` expressions are also derived expressions.

3 Evaluator Data Structures

In addition to defining the external syntax of expressions, the evaluator implementation must also define the data structures that the evaluator manipulates internally, as part of the execution of a program, such as the representation of procedures and environments and the representation of true and false.

3.1 Testing of predicates

For conditionals, we accept anything to be true that is not the explicit `false` object.

```
(define (true? x)
  (not (eq? x false)))

(define (false? x)
  (eq? x false))
```

3.2 Representing procedures

To handle primitives, we assume that we have available the following procedures:

- `(apply-primitive-procedure <proc> <args>)` applies the given primitive procedure to the argument values in the list *<args>* and returns the result of the application.
- `(primitive-procedure? <proc>)` tests whether *<proc>* is a primitive procedure.

  These mechanisms for handling primitives are further described in section 4.1.4.

  Compound procedures are constructed from parameters, procedure bodies, and environments using the constructor `make-procedure`:

```
(define (make-procedure parameters body env)
  (list 'procedure parameters body env))

(define (compound-procedure? p)
  (tagged-list? p 'procedure))
```

```scheme
(define (procedure-parameters p) (cadr p))

(define (procedure-body p) (caddr p))

(define (procedure-environment p) (cadddr p))
```

3.3 Operations on Environments

The evaluator needs operations for manipulating environments. As explained in section 3.2, an environment is a sequence of frames, where each frame is a table of bindings that associate variables with their corresponding values. We use the following operations for manipulating environments:

- `(loopup-variable-value <var> <env>)` returns the value that is bound to the symbol *<var>* in the environment *env*, or signals an error if the variable if unbound.

- `(extend-environment <variables> <values> <base-env>)` returns a new environment, consisting of a new frame in which the symbols in the list *<variables>* are bound to the corresponding elements in the list *<values>*, where the enclosing environment is the environment *<base-env>*.

- `(define-variable!  <var> <value> <env>)` adds to the first frame in the environment *<env>* a new binding that associates the variable *<var>* with the value /<value».

- `(set-variable-value!  <var> <value> <env>)` changes the binding of the variable *<var>* in the environment *<env>* so that the variable is now bound to value *<value>*, or signals an error if the variable is unbound.

  To implement these operations we represent an environment as a list of frames. The enclosing environment of an environment is the `cdr` of the list. The empty environment is simply an empty list.

```scheme
(define (enclosing-environment env) (cdr env))

(define (first-frame env) (car env))

(define the-empty-environment '())
```

Each frame of an environment is represented as a pair of lists: a list of the variables bound in that frame and a list of the associated values.

```
(define (make-frame variables values)
  (cons variables values))

(define (frame-variables frame) (car frame))

(define (frame-values frame) (cdr frame))

(define (add-binding-to-frame! var val frame)
  (set-car! frame (cons var (car frame)))
  (set-cdr! frame (cons val (cdr frame))))
```

To extend an environment by a new frame that associates variables with values, we make a frame consisting of the list of variables and the list of values, and we adjoin this to the environment. We signal an error if the number of variables does not match the number of values.

```
(define (extend-environment vars vals base-env)
  (if (= (length vars) (length vals))
      (cons (make-frame vars vals) base-env)
      (if (< (length vars) (length vals))
          (error "Too many arguments supplied" vars vals)
          (error "Too few arguments supplied" vars vals))))
```

To look up a variable in an environment, we scan the list of variables in the first frame. If we find the desired variable, we return the corresponding element in the list of values. If we do not find the variable in the current frame, we search the enclosing environment, and so on. If we reach the empty environment, we signal an "unbound variable" error.

```
(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))
```

To set a variable to a new value in a specified environment, we scan for the variable, just as in `lookup-variable-value`, and change the corres value when we find it.

```
(define (set-variable-value! var val env)
  (define (env-loop env)
    (define (scan vars vals)
      (cond ((null? vars)
             (env-loop (enclosing-environment env)))
            ((eq? var (car vars))
             (set-car! vals val))
            (else (scan (cdr vars) (cdr vals)))))
    (if (eq? env the-empty-environment)
        (error "Unbound variable -- SET!" var)
        (let ((frame (first-frame env)))
          (scan (frame-variables frame)
                (frame-values frame)))))
  (env-loop env))
```

To define a variable, we search the first frame for a binding for the variable, and change the binding if it exists (just as in `set-variable-value!`). If no such binding exists, we adjoin one to the first frame.

```
(define (define-variable! var val env)
  (let ((frame (first-frame env)))
    (define (scan vars vals)
      (cond ((null? vars)
             (add-binding-to-frame! var val frame))
            ((eq? var (car vars))
             (set-car! vals val))
            (else (scan (cdr vars) (cdr vals)))))
    (scan (frame-variables frame)
          (frame-values frame))))
```

4 Running the Evaluator as a Program

Given the evaluator, we have in our hands a description (expressed in Lisp) of the process by which Lisp expressions are evaluated. One advantage of expressing the evaluator as a program is that we can run the program. This gives up, running with Lisp, a working model of how Lisp itself evaluates expressions. This can serve as a framework for experimenting with evaluation rules, as we shall do later in this chapter.

Our evaluator program reduces expressions ultimately to the application of primitive procedures. Therefore, all that we need to run the evaluator is to create a mechanism that calls on the underlying Lisp system to model the application of primitive procedures.

There must be a binding for each primitive procedure name, so that when `eval` evaluates the operator of an application of a primitive, it will find an object to pass to `apply`. We thus set up a global environment that associates unique objects with the names of the primitive procedures that can appear in the expressions we will be evaluating. The global environment also includes bindings for the symbols `true` and `false`, so that they can be used as variables in expressions to be evaluated.

```
(define (setup-environment)
  (let ((initial-env
         (extend-environment (primitive-procedure-names)
                             (primitive-procedure-objects)
                             the-empty-environment)))
    (define-variable! 'true true initial-env)
    (define-variable! 'false false initial-env)
    initial-env))
```

It does matter how we represent the primitive procedure objects, so long as `apply` can identify and apply them by using the procedures `primitive-procedure?` and `apply-primitive-procedure`. We have chosen to represent a primitive procedure as a list beginning with the symbol `primitive` and containing a procedure in the underlying Lisp that implements that primitive.

```
(define (primitive-procedure? proc)
  (tagged-list? proc 'primitive))

(define (primitive-implementation proc) (cadr proc))
```

`Setup-environment` will get the primitive names and implementation procedures from a list:

```scheme
;; TODO add more
(define primitive-procedures
  (list (list 'car car)
        (list 'cdr cdr)
        (list 'cons cons)
        (list 'null? null?)))

(define (primitive-procedure-names)
  (map car primitive-procedures))

(define (primitive-procedure-objects)
  (map (lambda (proc) (list 'primitive (cadr proc)))
  ↪  primitive-procedures))
```

To apply a primitive procedure, we simply apply the implementation procedure to the arguments, using the underlying Lisp system:

```scheme
(define (apply-primitive-procedure proc args)
  (apply-in-underlying-scheme
   (primitive-implementation proc) args))
```

For convenience in running the metacircular evaluator, we provide a *driver loop* that models the read-eval-print loop of the underlying Lisp system.

```scheme
(define input-prompt ";; M-Eval input:")
(define output-prompt ";; M-Eval value:")

(define (prompt-for-input str)
  (newline) (newline) (display str) (newline))

(define (announce-output str)
  (newline) (display str) (newline))

(define (user-print object)
  (if (compound-procedure? object)
      (display (list 'compound-procedure
                     (procedure-parameters object)
                     (procedure-body object)
                     '<procedure-env>))
      (display object)))
```

```scheme
(define (driver-loop)
  (prompt-for-input input-prompt)
  (let ((input (read)))
    (let ((output (eval input the-global-environment)))
      (announce-output output-prompt)
      (user-print output)))
  (driver-loop))

;; This has to be put before defining the metacircular apply
(define apply-in-underlying-scheme apply)

;(define the-global-environment (setup-environment))
;(driver-loop)
```

5 source code

```scheme
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotation exp))
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-definition exp env))
        ((if? exp) (eval-if exp env))
        ((lambda? exp)
         (make-procedure (lambda-parameters exp)
                         (lambda-body exp)
                         env))
        ((begin? exp)
         (eval-sequence (begin-actions exp) env))
        ((cond? exp) (eval (cond->if exp) env))
        ((application? exp)
         (apply (eval (operator exp) env)
                (list-of-values (operands exp) env)))
        (else
         (error "Unknown expression type -- EVAL" exp))))
;; This has to be put before defining the metacircular apply
(define apply-in-underlying-scheme apply)
(define (apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
         (eval-sequence
          (procedure-body procedure)
          (extend-environment
           (procedure-parameters procedure)
           arguments
           (procedure-environment procedure))))
        (else
         (error
          "Unknown procedure type -- APPLY" procedure))))
(define (list-of-values exps env)
  (if (no-operands? exps)
      '()
      (cons (eval (first-operand exps) env)
            (list-of-values (rest-operands exps) env))))
(define (eval-if exp env)
  (if (true? (eval (if-predicate exp) env))
      (eval (if-consequent exp) env)
      (eval (if-alternative exp) env)))
(define (eval-sequence exps env)
  (cond ((last-exp? exps) (eval (first-exp exps) env))
        (else (eval (first-exp exps) env)
              (eval-sequence (rest-exps exps) env))))
(define (eval-assignment exp env)
  (set-variable-value! (assignment-variable exp)
                       (eval (assignment-value exp) env)
                       env)
  'ok)
(define (eval-definition exp env)
  (define-variable! (definition-variable exp)
                    (eval (definition-value exp) env)
                    env)
  'ok)
(define (self-evaluating? exp)
  (cond ((number? exp) true)
        ((string? exp) true)
        (else false)))
(define (variable? exp) (symbol? exp))
(define (quoted? exp)
  (tagged-list? exp 'quote))

(define (text-of-quotation exp) (cadr exp))
(define (tagged-list? exp tag)
  (if (pair? exp)
      (eq? (car exp) tag)
      false))
(define (assignment? exp)
  (tagged-list? exp 'set!))

(define (assignment-variable exp) (cadr exp))

(define (assignment-value exp) (caddr exp))
(define (definition? exp) (tagged-list? exp 'define))

(define (definition-variable exp)
  (if (symbol? (cadr exp))
      (cadr exp)
      (caadr exp)))

(define (definition-value exp)
  (if (symbol? (cadr exp))
      (caddr exp)
      (make-lambda (cdadr exp)   ;formal parameters
                   (cddr exp)))) ;body
(define (lambda? exp) (tagged-list? exp 'lambda))

(define (lambda-parameters exp) (cadr exp))

(define (lambda-body exp) (cddr exp))

(define (make-lambda parameters body)
  (cons 'lambda (cons parameters body)))
(define (if? exp) (tagged-list? exp 'if))

(define (if-predicate exp) (cadr exp))

(define (if-consequent exp) (caddr exp))

(define (if-alternative exp)
  (if (not (null? (cdddr exp)))
      (cadddr exp)
      'false))

(define (make-if predicate consequent alternative)
  (list 'if predicate consequent alternative))
(define (begin? exp) (tagged-list? exp 'begin))

(define (begin-actions exp) (cdr exp))

(define (last-exp? seq) (null? (cdr seq)))

(define (first-exp seq) (car seq))

(define (rest-exps seq) (cdr seq))

(define (make-begin seq) (cons 'begin seq))

(define (sequence->exp seq)
  (cond ((null? seq) seq)
        ((last-exp? seq) (first-exp seq))
        (else (make-begin seq))))
(define (sequence->exp seq)
  (cond ((null? seq) seq)
        ((last-exp? seq) (first-exp seq))
        (else (make-begin seq))))

(define (make-begin seq) (cons 'begin seq))
(define (application? exp) (pair? exp))

(define (operator exp) (car exp))

(define (operands exp) (cdr exp))

(define (no-operands? ops) (null? ops))

(define (first-operand ops) (car ops))

(define (rest-operands ops) (cdr ops))
```

```
(define (cond? exp) (tagged-list? exp 'cond))

(define (cond-clauses exp) (cdr exp))

(define (cond-predicate clause) (car clause))

(define (cond-actions clause) (cdr clause))

(define (cond-else-clause? clause)
  (eq? (cond-predicate clause) 'else))

(define (cond->if exp)
  (expland-clauses (cond-clauses exp)))

(define (expand-clauses clauses)
  (if (null? clauses)
      'false
      (let ((first (car clauses))
            (rest (cdr clauses)))
        (if (cond-else-clause? first)
            (if (null? rest)
                (sequence->exp (cond-actions first))
                (error "ELSE clause isn't last -- COND->IF"
                       clauses))
            (make-if (cond-predicate first)
                     (sequence->exp (cond-actions first))
                     (expand-clauses rest))))))
(define (true? x)
  (not (eq? x false)))

(define (false? x)
  (eq? x false))
(define (make-procedure parameters body env)
  (list 'procedure parameters body env))

(define (compound-procedure? p)
  (tagged-list? p 'procedure))

(define (procedure-parameters p) (cadr p))
```

```
(define (procedure-body p) (caddr p))

(define (procedure-environment p) (cadddr p))

(define (setup-environment)
  (let ((initial-env
         (extend-environment (primitive-procedure-names)
                             (primitive-procedure-objects)
                             the-empty-environment)))
    (define-variable! 'true true initial-env)
    (define-variable! 'false false initial-env)
    initial-env))
(define (primitive-procedure? proc)
  (tagged-list? proc 'primitive))

(define (primitive-implementation proc) (cadr proc))
;; TODO add more
(define primitive-procedures
  (list (list 'car car)
        (list 'cdr cdr)
        (list 'cons cons)
        (list 'null? null?)))

(define (primitive-procedure-names)
  (map car primitive-procedures))

(define (primitive-procedure-objects)
  (map (lambda (proc) (list 'primitive (cadr proc)))
   ↪  primitive-procedures))
(define (apply-primitive-procedure proc args)
  (apply-in-underlying-scheme
   (primitive-implementation proc) args))

(define input-prompt ";; M-Eval input:")
(define output-prompt ";; M-Eval value:")

(define (prompt-for-input str)
  (newline) (newline) (display str) (newline))

(define (announce-output str)
```

```scheme
  (newline) (display str) (newline))

(define (user-print object)
  (if (compound-procedure? object)
      (display (list 'compound-procedure
                     (procedure-parameters object)
                     (procedure-body object)
                     '<procedure-env>))
      (display object)))

(define (driver-loop)
  (prompt-for-input input-prompt)
  (let ((input (read)))
    (let ((output (eval input the-global-environment)))
      (announce-output output-prompt)
      (user-print output)))
  (driver-loop))
```