Meta-Circular Interpreter

A Scheme interpreter written in Scheme.
(See SICP: Ch 4 or 341 Winter 1999 Homework 3)

Why?

- Tweaking language, trying new constructs.
- Debugging.
- Because we can (and it is easy)!
- Educational value.

MCI: internals

Eval: `(eval <expr> <env>)` evaluates S-Expression `<expr>` in environment `<env>`.

Apply: `(apply <proc> <arg-list>)` does procedure applications.

Environment: List of environment frames.

Environment Frame: List of variable bindings.
E.g. `((x 10) (y "hi"))`

Procedures: Tuples of `(<param-list> <body> <env>)`.

Read-Eval-Print Loop

```
(define the-global-env (...))

(define (read-eval-print)
   (display "MCI> ")
   (display (eval (read) the-global-env))
   (newline)
   (read-eval-print)
)
```


Environments

Recall evaluation procedure for symbols:

- Look up symbol's value in the *current environment*.
- Return the value.

So, what is the current environment?

At the *read-eval-print* loop prompt it is the global environment.

What about inside a function or a let?

Environments (cont)

An *environment frame* is a set of variable bindings:

```
((x 10) (y 20) (+ #<primitive-procedure +>) ...)
```

The *environment* is a list of environment frames.

To look up `x` we first check each frame until we find it.

The Global Environment

Contains only the global environment frame:

```
((+ #<primitive-procedure +>)
 (- #<primitive-procedure ->)
 ...
 (car #<primitive-procedure car>))
```

After `(define x 10)` we have:

```
((x 10)
 (+ #<primitive-procedure +>)
 (- #<primitive-procedure ->)
 ...
 (car #<primitive-procedure car>))
```


`let` Environments

In what environment do we evaluate `(+ x y)` in:

```
(let ((x 10)
      (y 100)
     )
     (+ x y)
)
```


`let` Environments (cont)

```
(((x 10)
  (y 100))
 ((+ #<primitive-procedure +>)
  (- #<primitive-procedure ->)
  ...
  (car #<primitive-procedure car>)))
```

`eval`

Scheme code for evaluation procedure discussed in first Scheme lecture.

```
(define (eval expr env)
   (cond ((symbol? expr) (lookup env symbol))
         ((literal? expr) expr)
         ((special-form? expr) (eval-special-form expr env))
         ((pair? expr)
             (let ((exp-list (map (lambda (arg) (eval arg env)) expr)))
                (apply (car exp-list) (cdr exp-list))
             )
         )
   )
)
```


Lexical Scoping

(also called static scoping)

Variables are bound when procedures are defined not when they are called. For example:

```
(define x 10)
(define (x-plus y) (+ x y))

(x-plus 2)

(define (x-plus-double x) (x-plus (* 2 x)))

(x-plus-double 2)
```

In a procedure application, the *free variables* in the body of a procedure are evaluated in the procedure's *defining environment*.

A procedure is ...

Represented by tuple consisting of:

- A parameter list.
- A body.
- The procedure's defining environment.

MCI procedure selectors: `(proc-params proc)`, `(proc-body proc)`, and `(proc-env proc)`.

Applying Procedures

Scheme primitive `(apply <proc> <arg-list>)`
For Example: `(+ 1 2 3)` is same as `(apply + '(1 2 3))`

For Meta-Circular Interpreter:

```
(define (apply proc args)
   (let ((new-frame (map list (proc-params proc) args)))
        (eval (proc-body proc) (cons new-frame (proc-env proc)))
   )
)
```

Defining Environments

Consider:

```
guile> (define z 10)
guile> (define foo (lambda (x y) (+ x y z)))

guile> (define bar (let ((z 100)) (lambda (x y) (+ x y z))))
```


Eval `(foo 2 3)`

Add local environment frame,

```
((x 2) (y 3))
```

to defining environment to get:

```
(((x 2)
  (y 3))
 ((bar #<procedure bar (x y)>)
  (foo #<procedure foo (x y)>)
  (z 10)
  (+ #<primitive-procedure +>)
  (- #<primitive-procedure ->)
  ...
  (car #<primitive-procedure car>)))
```

Evaluate body `(+ x y z)` in this environment.

Eval `(bar 2 3)`

Add local environment frame,

```
((x 2) (y 3))
```

to defining environment to get:

```
(((x 2)
  (y 3))
 ((z 100))
 ((bar #<procedure bar (x y)>)
  (foo #<procedure foo (x y)>)
  (z 10)
  (+ #<primitive-procedure +>)
  (- #<primitive-procedure ->)
  ...
  (car #<primitive-procedure car>)))
```

Evaluate body `(+ x y z)` in this environment.

`lambda` and `let`

Recall how procedure application works:

```
(<func> <arg-1> <arg-2> ... )
```

Here `<func>` is just an S-expression that evaluates to a procedure. E.g.

```
((if #f car cdr) '(1 2 3 4))

(define z 10)
(define w -5)
((lambda (x) (* x x)) (+ z w))
```

lambda and let (cont)

Compare:

```
(define z 10)
(define w -5)

((lambda (x) (* x x)) (+ z w))
```

with

```
(let ((x (+ z w)))
     (* x x)
)
```

`let` is not so special. It is a source-to-source transformation to `lambda`.

Nested lets: let*

Derived form: `let*`

```
(let* ((<var-1> <expr-1>)          ⇒   (let ((<var-1> <expr-1>))
       (<var-2> <expr-2>)                   (let ((<var-2> <expr-2>))
       ...                                     ...
       (<var-n> <expr-n>)                       (let ((<var-n> <expr-n>))
      )                                              <body>
      <body>                                    )
)                                           )
                                       )
```


Extended Example

```
(define (my-map func lst)
   (if (null? lst)
       '()
       (cons (func (car lst))
             (my-map func (cdr lst)))))

(define (const-func val)
   (lambda (x) val))

(define const1 (const-func 1))
(define const5 (const-func 5))

(my-map const1 '(1 2 3))
(my-map const5 '(1 2 3))
```


Tail Recursive foldl

Recursive helper function is defined locally not globally:

```
(define (foldl op id lst)
   (define (foldl-helper val lst)
      (if (null? lst)
          val
          (foldl-helper (op val (car lst)) (cdr lst))
      )
   )
   (foldl-helper id lst)
)
```

Note how lexical scoping is used for `op` in `foldl-helper`.

More Glue

Define `compose` to take two single argument functions f and g and return a function h that is f composed with g. That is

$$h(x) = f(g(x))$$

For example:

```
(define my-cadr (compose car cdr))
(define my-cadaar (foldr compose identity (list car cdr car car)))
```


More Glue (cont)

Define `rev2` that takes a two argument function f and returns a new two argument function g that takes the arguments in reverse. That is:

$$h(x, y) = f(y, x)$$

```
(define (reverse lst)
   (foldl (rev2 cons) '() lst)
)
```