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 expr))

         ((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 (in our MCI):

- 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 implemented with a *source-to-source transformation* to `lambda`. `let` is a *derived form*.

Nested `let`s: `let*`

Derived form: `let*`

⇒

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

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


Variable Argument Lengths

Scheme has primitive functions `+`, `list`, `map`, etc. that accept a variable number of arguments. E.g.

```
(list)              => ()
(list 1)            => (1)
(list 1 2)          => (1 2)
```

Can we define our own?

You Bet!

These are identical:

```
(define (my-list . args) args)

(define my-list (lambda args args))
```

Note this “dot” notation is the same thing you get when you `cons` something to a non-list.

What if we want to require some arguments?

You Bet! (cont)

```
(define (my-map func . lists) ...)

(define my-map (lambda (func . lists) ...))
```


Bootstrapping

Creating something from a simpler version. For Example:

```
(define (my-map1 func lst)
   (if (null? lst)
       '()
       (cons (func (car lst)) (my-map1 func (cdr lst)))
   )
)

(define (my-map func . lists)
   (if (null? (car lists))
       '()
       (cons (apply func (my-map1 car lists))
             (apply my-map (cons func (my-map1 cdr lists)))
       )
   )
)
```


The Power of Lambda

Look for powerful constructs in a language.

Lambda is powerful.

We have already seen that it can be used in place of `let`.

What else can it do?

`cons` cells

Scheme has primitive procedures `cons`, `car`, and `cdr`.

Can we get by without them? Yes! Define abstract `cons` cell using `lambda` for implementation:

```
(define (my-cons the-car the-cdr)
   (lambda (which)
      (cond ((eq? which 'car) the-car)
            ((eq? which 'cdr) the-cdr)
            (else (error "cons cell usage error"))
      )
   )
)

(define (my-car the-pair)
   (the-pair 'car)
)

(define (my-cdr the-pair)
   (the-pair 'cdr)
)
```


Manipulating Functions

We have seen:

- Function creation (static).
- Function application (static and dynamic).
- Functions in variables.

We can also:

- Manipulate functions to dynamically create new functions (E.g. `const-func`).

Dynamic Function Creation

Facilitated by:

- `lambda` to create new functions.
- First-class-ness of functions.
- *Persistent* environments.

Currying

Named for Haskell B. Curry

- Given n argument function f.
- Specify k arguments to get $n-k$ argument function g.
- If $g = \text{curry}(f, a_1, \ldots, a_k)$ then $g(a_{k+1}, \ldots, a_n) = f(a_1, \ldots, a_n)$

Currying (cont)

A `curry` for a two argument function, f:

```
(define (curry2 f x)
   (lambda (y) (f x y))
)
```

But we can use variable argument lengths to write a general `curry`:

```
(define (curry f . first-args)
   (lambda second-args (apply f (append first-args second-args)))
)
```


Uses for `curry`

Lambda definitions are hard to read, curried functions are more clear.

```
(define add-lists (curry map +))
(add-lists '(1 2 3) '(4 5 6))

(map (curry + 5) '(1 2 3))

(define positive-only (curry filter positive?))

(define sum (curry foldr + 0))
(define sum (curry apply +))
```


Composing

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 (identity x) x)

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


Reversing Arguments

Define `reverse-args` that takes a function f and returns a new function g that takes the arguments in reverse. That is:

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

How about a more general version?

More Examples

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


;; from eval definition
(map (lambda (arg) (eval arg env)) expr)
;; becomes
(map (curry (reverse-args eval) env) expr)
```


Building Blocks

We have seen some useful higher-order building blocks: `map`, `foldr`, `curry`, `compose`, and `reverse-args`.

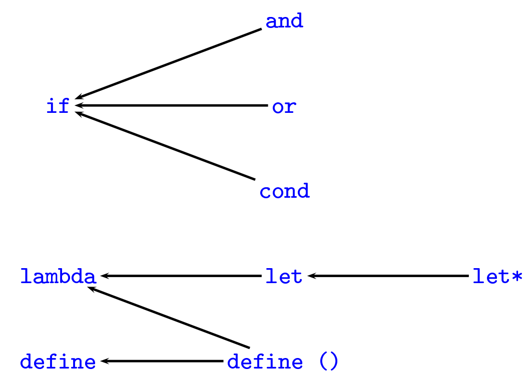
Scheme could provide these.

Instead Scheme provides the equipment to make them and more.

Derived Expressions

(aka Source-to-Source transformations)
These are special-forms that can be stated in terms of other special-forms.

Redundant in that Scheme is equally as expressive without them.



let->combination

Turn a symbolic `let` expression into a `lambda` and application.
For example:

```
guile> (let->combination '(let ((x (+ 5 6)) (y 10)) (+ x y)))
((lambda (x y) (+ x y)) (+ 5 6) 10)

(define (let->combination expr)
   (let ((bindings (cadr expr))
         (body     (cddr expr))
        )
        (let ((vars (map car bindings))
              (exps (map cadr bindings))
             )
             (cons (cons 'lambda (cons vars body)) exps)
        )
   )
)
```

Recall 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>                                 )
)                                       )
                                   )
```


and->if

Convert `(and <e1> <e2> <e3>)` to an `if` expression

Recall Lexical Scoping

Free 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)                        => 12

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

(x-plus-double 2)                 => 14
```


Their Environments

The environment where `(* 2 x)` is evaluated is:

```
(((x 2))
 ((x 10)
  (+ #<primitive-procedure +>)
  (- #<primitive-procedure +>)
  ...
 ))
```

And we get 4.

The environment where `(+ x y)` is evaluated is:

```
(((y 4))
 ((x 10)
  (+ #<primitive-procedure +>)
  (- #<primitive-procedure +>)
  ...
 ))
```

And we get 14.

Dynamic Scoping

The *free variables* are bound *dynamically* at procedure call.
Local environment frame is added to calling procedure's environment.

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

(x-plus 2)                    => 12

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

(x-plus-double 2)             => 6
```

Environment frames in dynamic scoping correspond with the call stack.

Their Environments

The environment where `(* 2 x)` is evaluated is:

```
(((x 2))
 ((x 10)
  (+ #<primitive-procedure +>)
  (- #<primitive-procedure +>)
  ...
 ))
```

And we get 4.

The environment where `(+ x y)` is evaluated is:

```
(((y 4))
 ((x 2))
 ((x 10)
  (+ #<primitive-procedure +>)
  (- #<primitive-procedure +>)
  ...
 ))
```

And we get 6.

Why Dynamic Scoping?

- It is a way to pass parameters.
- Unix environment variables use dynamic scoping.
- *Exceptions handlers* are dynamically scoped.

Scheme Programs

Want to execute your scheme code as a program? Write "shell" scripts in Scheme?

```
#!/usr/bin/guile -s
!#

;;; this shell script prints out its
;;; arguments in reverse order

(for-each              ;; like 'map', but ignores value.
   (lambda (x)
      (display x)
      (newline)
   )
   (reverse (command-line))
)
```

Where is Scheme?

- Education: used for introductory programming courses.
- Artificial Intelligence.
- Application Extension Language.
- Rapid Prototyping.

Scheme Summary

- Expression oriented.
- Uses lists extensively.
- Uses recursive extensively, has tail recursive optimization.
- Lexically scoped.
- Functional style (not pure).
- Interpreted, but compilers exist.
- Simple syntax but expressive.
- Dynamically typed (and type-safe).
- Garbage collected.