APP Autumn24 Lecture 33 (Tuesday, Oct 29, 2024)

Let us design an interpreter for Scheme. For simplicity, we would skip several of the special forms; as there aren't many in Scheme anyway, adding them wouldn't be much non-trivial.

Recall the evaluation rule: we evaluate an expression by evaluating the subexpressions corresponding to the operator and the operands, and then apply the procedure represented by the operator on to the operands. As you can see, evaluating an expressions involves evaluating its subexpressions, so like all our other interesting Scheme programs, our evaluator or interpreter will be a recursive program; and again like most of our interesting recursive programs, the interpreter will be a case analysis, the cases here being the kind of the expression being evaluated. You would notice that what we write today can be called the "kernel" of the interpreters of many more languages -- complex or simple -- as the underlying idea, of a case analysis using an environment model of evaluation, is common for most programming languages around.

So here is our main procedure 'eval':

```
(define eval
  (lambda (exp env)
    (cond ( ... ))))
```

- Let us start with numbers. Say the interpreter gets a '4', what should it give as the evaluated result? A '4'.

```
((number? exp) exp)
```

- What if it's a symbol like 'x', 'foo'? We look up its value in the environment:

```
((symbol? exp) (lookup exp env))
```

- What about quoted expressions? Say 'foo is represented as (quote foo) -- equivalent in Scheme. We want the quoted expression as it is:

```
((eq? (car exp) 'quote) (cadr exp))
```

- What else? Lambdas? We create a procedure object, or a closure, which contains the list of parameters, the procedure body, and the current environment.

```
(lambda (x y) (+ x y)) ==> (closure ((x y) (+ x y) env))

((eq? (car exp) 'lambda) (list 'closure (cadr exp) (cddr exp)))
```

- Conditionals? Say 'if' can be represented as 'cond'.

```
(cond (p1 e1)
      (p2 e2)
      (else en))

((eq? (car exp) 'cond')
   (evalcond (cdr exp) env))
```

// Notice that like 'evalcond' we could have "functionalized" other actions as well.

What is the last one (without that you can't do anything actually!)? Procedure application.

(proc op1 op2 ...) ==> Evaluate "proc", "op1", "op2", etc. and apply the procedure represented by "proc" to the arguments "op1", "op2", etc. in a new environment formed by extending the environment in the closure of "proc" with the bindings for its parameters.

```
(else (apply (eval (car exp) env)
             (evallist (cdr exp) env)))
```

Was there something you could have checked for before deeming the above case as a procedure application? Basically you need at least two things, right, a procedure and a list of arguments. So we could have done better by checking the following:

```
((pair? exp) (apply (eval (car exp) env)
             (evallist (cdr exp) env)))
(else (error "unknown expression type"))
```

That's it. Our Scheme is a really small yet powerful language! Remarkably, our interpreter itself is so simple that it doesn't even use the powerful features of Scheme such as higher order functions!!

// Now let's complete the subsidiary definitions.

- The most important one is "apply":

```
(define (apply
  (lambda (proc args)
    (cond ((...))))))
```

Again a case statement, implying an interesting procedure!

- What's the simplest kind of application? A primitive one:

```
((primitive-op? proc)
  (apply-prim-op proc args))
```

where "apply-prim-op" say just executes the machine instructions corresponding to "procedure" in the global environment.

- How do we apply a compound procedure? Repeated enough times yesterday and today!

```
((compound-op? proc)
  (...)
```

// Before we proceed, a coffee for someone who can tell the definition of "compound-op"? (eq? (car proc) 'closure')

```
((compound-op? proc)
  (eval (caddr proc)
        (extend-environ (cadr proc) args (cadddr proc))))
```

- Here are the last two:

```
(define evallist
  (lambda (args env)
    (cond ((eq? args '()) '())
          (else
            (cons (eval (car l) env)
                  (evallist (cdr l) env))))))

(define evalcond
  (lambda (clauses env)
    (cond ((eq? clauses '()) '())
          ((eq? (caar clauses) 'else)
             (eval (cadar clauses) env))
          ((false? (eval (caar clauses) env))
             (evalcond (cdr clauses) env))
          (else
            (eval (cadar clauses) env)))))
```

SUMMARY:

```
(lambda (x y) (+ x y))

(define eval
```

```
  (lambda (exp env)
    (cond ((number? exp) exp)
          ((symbol? exp) (lookup exp env))
          ((eq? (car exp) 'quote) (cadr exp))
          ((eq? (car exp) 'lambda) (list 'closure (cadr exp) (cddr exp)
env))
          ((eq? (car exp) 'cond) (evalcond (cdr exp) env))
          ((pair? exp) (apply (eval (car exp) env) (evallist (cdr exp)
env)))
          (else (error "unknown expression type")))))

(define (apply
  (lambda (proc args)
    (cond ((primitive-op? proc) (apply-prim-op proc args))
          ((compound-op? proc) (eval (caddr proc) (extend-environ (cadr
proc) args (cadddr proc)))))))))

(define evallist
  (lambda (args env)
    (cond ((eq? args '()) '())
          (else
            (cons (eval (car l) env)
                  (evallist (cdr l) env))))))

(define evalcond
  (lambda (clauses env)
    (cond ((eq? clauses '()) '())
          ((eq? (caar clauses) 'else)
             (eval (cadar clauses) env))
          ((false? (eval (caar clauses) env))
             (evalcond (cdr clauses) env))
          (else
            (eval (cadar clauses) env)))))
```

- We still haven't defined "extend-environ" and "lookup", but they can be implemented simply as tables or a map of key-value pairs, which I assume you would find straightforward yet educative to implement yourselves.

Example:

```
(eval '(((lambda (x) (lambda (y) (+ x y))) 3) 4) <e0>)

(apply (eval '((lambda (x) (lambda (y) (+ x y))) 3) <e0>)
       (evallist '(4) <e0>))

(apply (eval '((lambda (x) (lambda (y) (+ x y))) 3) <e0>)
       (cons (eval '4 <e0>) (evallist '() <e0>)))

(apply (eval '((lambda (x) (lambda (y) (+ x y))) 3) <e0>)
       '(4))
```

```
(apply (apply (eval '(lambda (x) (lambda (y) (+ x y))) <e0>)
              '(3))
       '(4))

(apply (apply '(closure (x) (lambda (y) (+ x y)) <e0>)
              '(3))
       '(4))

(apply (eval '(lambda (y) (+ x y)) <e1>)
       '(4))

(apply '(closure (y) (+ x y) <e1>)
       '(4))

(eval '(+ x y) <e2>)

(apply (eval '+ <e2>) (evallist (x y) <e2>))

(apply <+-proc> '(3 4))

7
```

Final notes

- Could we improve our implementation? Definitely, in many ways. Few of them being:

    1. Functionalize all the actions.
    2. Generalize the "syntax extractors"; e.g., "(eq? (car exp) 'lambda')" could be replaced with a function ("lambda?"). A big advantage? We no longer would have to depend on the language's elements, and our interpreter's kernel could actually be shared across all the languages with these features.
    3. Dispatch on expression types instead of checking so many conditionals. For efficiency, we could create a hashtable indexed with expression types, or an array with individual indices from an enumerator about expression types, and directly jump to routines for each type.

- We still have not accounted for "defines" and "set!"s from Scheme, but handling them again is not so difficult -- you basically extend the map representing the environment. (An excellent mini project would be to implement this environment model for a given language subset of Scheme, but unfortunately there are so many existing implementations online, including the one in Scheme, that it would be very difficult to find if you really enjoyed the process or just used one of them. If you have enthu, do try writing it another programming language!)

- Over the next few classes, we shall see some tweaks to this implementation, realizing how small tweaks can lead to designing completely new languages and paradigms!

- Finally, notice the interesting cycle here: "eval" calls apply to simplify complex procedure applications (Procedure, Arguments), and "apply" calls eval to simplify subexpressions (Expression, Environment). This eval-apply cycle is the core of our interpreter, and that of most of the languages. Do you remember this quote from one of our physical classes?

"Any sufficiently complicated C or Fortran or ... program contains an ad hoc informally-specified bug-ridden slow implementation of half of Common Lisp."

That's true for Lisp's interpreter as well 😃

Thank you!

Provided Implementation

On Moodle you would also find a working "reference implementation" of the interpreter, with some of the added design elements described above. The way to use it in the DrRacket prompt is the following:

```
(myeval '(+ 2 3) the-global-environment)
```

Here, I would like you to play with the file and try to understand the following things:

1. The working of the interpreter, of course, along with the environment implementation.
2. How "the-global-environment" is populated.
3. Play with the interpreter to make it do different things than the default implementation!

We would see more interesting examples of Pt. 3 in the next classes.

Enjoy and Happy Diwali!