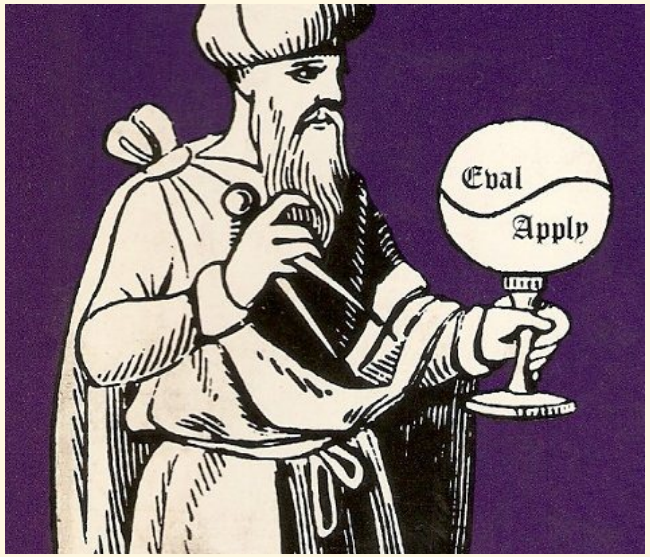


Write Your Own Lisp in Clojure

Making a tiny scheme interpreter

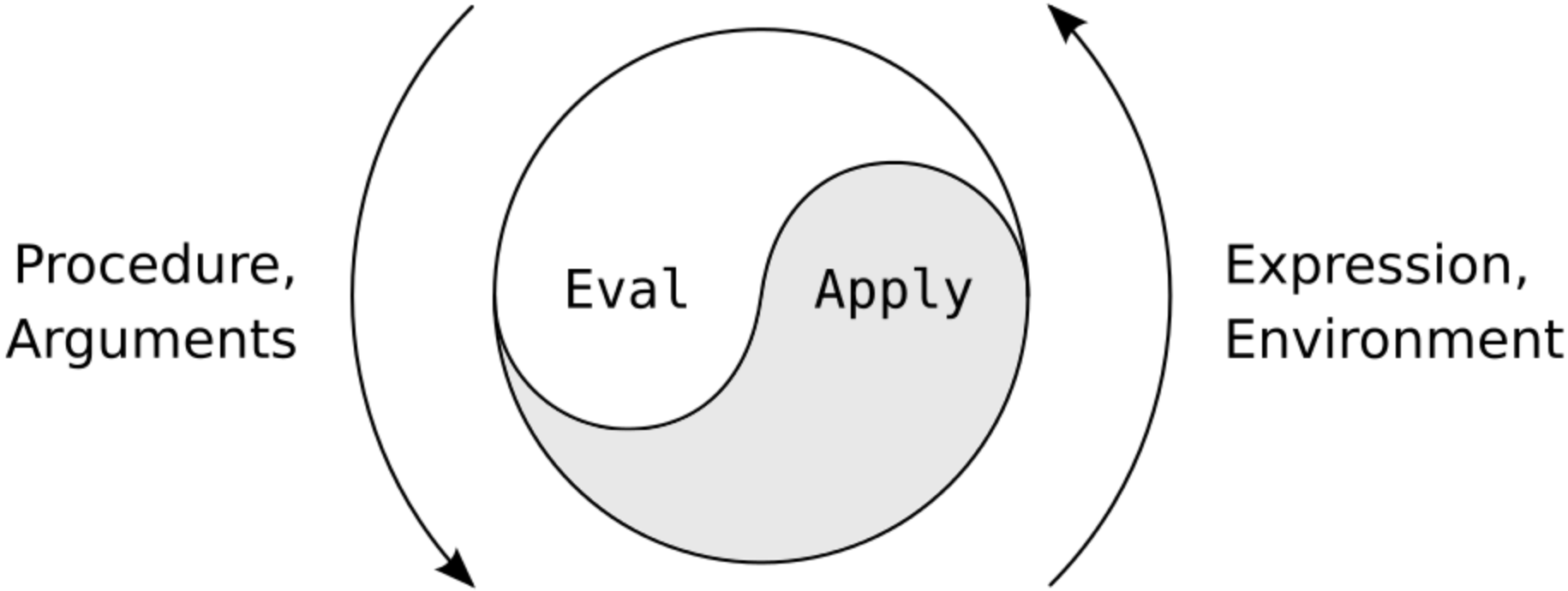
Who?

- Sam Lee (misgod.tw@gmail.com)
- work for Asus

An Interpreter

Demo

Eval & Apply



Eval

```
(defn form-eval [exp env]
  (let [exp (macroexpand exp env)]
    (cond (self-evaluating? exp)  exp
          (symbol? exp)           (env-get exp env)
          (= (first exp) 'quote)  (second exp)
          (= (first exp) 'if)     (eval-if exp env)
          (= (first exp) 'begin)  (eval-seq (rest exp) env)
          (= (first exp) 'lambda) (eval-lambda exp env)
          (= (first exp) 'define) (eval-define exp env)
          (= (first exp) 'set!)   (eval-assignment exp env)
          (= (first exp) 'let*)   (eval-let* exp env)
          (= (first exp) 'defmacro) (eval-defmacro exp env)
          (= (first exp) 'macroexpand) (macroexpand (second exp) env)
          :else (form-apply (form-eval (first exp) env)
                            (map #(form-eval % env) (rest exp))))))
```

Apply

```
(defn form-apply [proc args]
  (letfn [(primitive? [p] (= (first p) 'primitive))
          (apply-primitive [p a] (apply (second p) a)) ;;<-- magic
          (compound? [p] (= (first p) 'procedure))
          (proc-params [p] (second p))
          (proc-body [p] (nth p 2))
          (proc-env [p] (nth p 3))]
    (cond (primitive? proc) (apply-primitive proc args)
          (compound? proc)  (eval-seq (proc-body proc)
                                      (extend-env (proc-env proc)
                                                  (proc-params proc)
                                                  args)))))
```

Eval

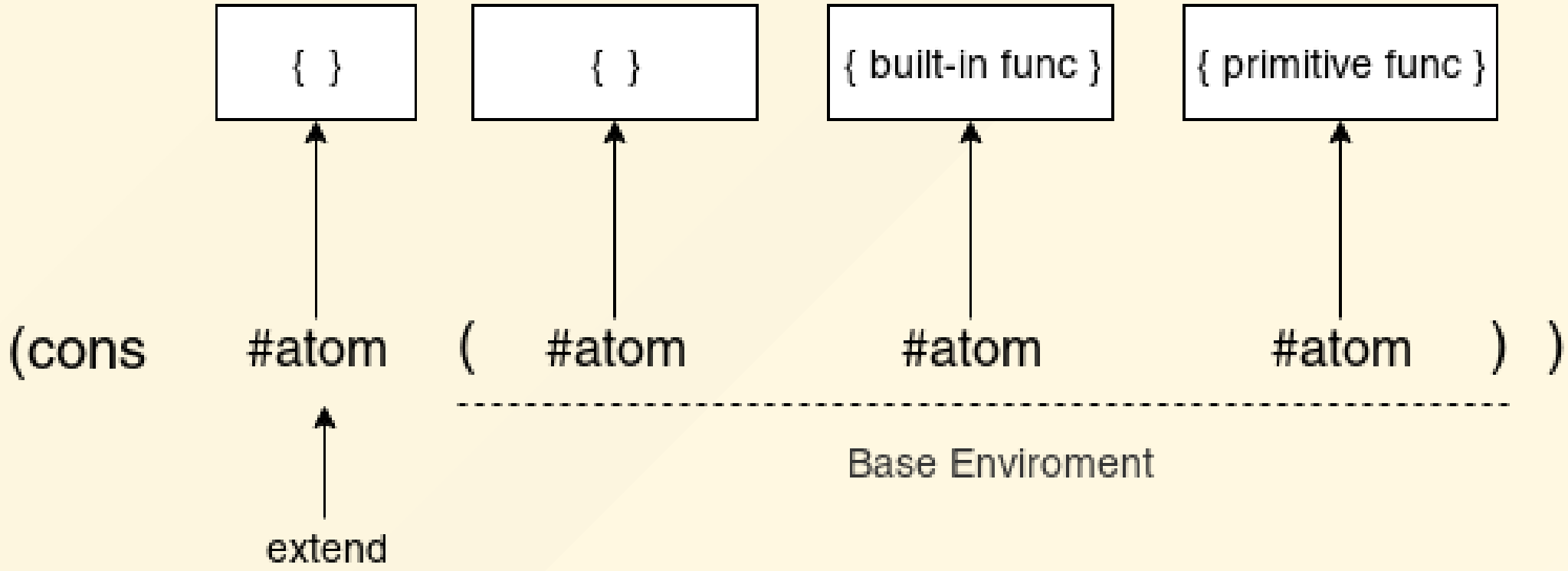
- Primitive expressions
  - self-evaluating expressions
  - variables
- Special forms
  - quoted expression
  - define, if, begin, lambda ...
- Combinations

Primitive expressions

```
(defn form-eval [exp env]
    (cond (self-evaluating? exp)  exp
          (symbol? exp) (env-get exp env)

(defn self-evaluating? [x]
    (or (number? x)
        (string? x)
        (nil? x)
        (instance? Boolean x)))
```

Enviroment


{ }
{ }
{ built-in func }
{ primitive func }
(cons
#atom
(
#atom
#atom
#atom
)
)
Base Enviroment
extend

Enviroment

```
(defn env-find [var env action not-found]  ...)

(defn env-get [var env] ...)

(defn env-set [var val env] ...)
```

Initial Enviroment

```
(defn setup-env []
  (-> '()
      (extend-env (keys primitive-procs)
                  (map #(list 'primitive %) (vals primitive-procs)))
      ;;above is base env
      (extend-env)))
```

```
(def primitive-procs {'true true
                      '+   +
                      '-   -
                      'car first
                      ...
                      })
```

quote

```
(defn form-eval [exp env]
    (cond (self-evaluating? exp)  exp
          (symbol? exp)           (env-get exp env)
          (= (first exp) 'quote)  (second exp)) ;;<- here
```

- (quote 1) => 1
- (quote a) => a
- (quote (+ 1 1)) => (+ 1 1)

if

```
(defn form-eval [exp env]
  (let [exp (macroexpand exp env)]
    (cond (self-evaluating? exp)  exp
          (symbol? exp)           (env-get exp env)
          (= (first exp) 'quote)  (second exp)
          (= (first exp) 'if)     (eval-if exp env)))) ; <- here

(defn eval-if [exp env]
  (let [[a0 a1 a2 a3] exp]
    (if (form-eval a1 env)
      (form-eval a2 env)
      (form-eval a3 env))))
```

- (if <cond-expr> expr else-expr)

Why if is a special form?

- Can we just write a if function?

```
(defn iif [condition stat else]
  (if (true? condition)
      expr
      else-expr))

(iif true (+ 1 2) (* 0 0)) ;=> 3

(iif false (+ 1 2) (* 0 0)) ;=> 0

(iif true (+ 1 2) (/ 0 0)) ;=> Error: ArithmeticException Divide by zero
```

begin

```
(defn form-eval [exp env]
    (cond (self-evaluating? exp)  exp
          (symbol? exp)           (env-get exp env)
          (= (first exp) 'quote)  (second exp)
          (= (first exp) 'if)     (eval-if exp env)
          (= (first exp) 'begin)  (eval-seq (rest exp) env))) ; <-- here
```

```
(defn eval-seq [exp env]
  (reduce #(form-eval %2 env) nil exp))
```

- like "do" in clojure

lambda

```
(defn form-eval [exp env]
  (cond (self-evaluating? exp)  exp
        ; ...
  (= (first exp) 'lambda) (eval-lambda exp env))))  ;<- here

(defn eval-lambda [exp env]
  (list 'procedure    ;<-- this is for form-apply
        (second exp)
        (drop 2 exp)
        env))
```

- (lambda (x y) (+ x y))
- (lambda () 5)

define

```
(defn form-eval [exp env]
  (let [exp (macroexpand exp env)]
    (cond (self-evaluating? exp)  exp
          ; ...
          (= (first exp) 'define) (eval-define exp env)))) ;<-- here
```

- (define x 1)
- (define (f x y) (+ x y))
- (define f (lambda (x y) (+ x y)))

eval-define

```
(defn define-var [exp]
    (if (seq? (second exp)) ;function?
      (first (second exp)) ;it's a function so the var is function name
      (second exp)))

(defn define-val [exp env]
  (let [var (second exp)
        make-lambda #(cons 'lambda  (cons %1 %2))]
    (if (seq? var) ;function?
      (form-eval (make-lambda (rest var) (drop 2 exp)) env)
      (form-eval (nth exp 2) env))))

(defn eval-define [exp env]
  (let [var (define-var exp)
        val (define-val exp env)]
    (swap! (first env) assoc var val)))
```

set!

```
(defn form-eval [exp env]
    (cond (self-evaluating? exp)  exp
          ;...
          (= (first exp) 'set!)   (eval-assignment exp env))) ;<-- here


(defn eval-assignment [exp env]
  (let [[op var val] exp]
    (env-set var val env)))
```

- (define x 5)
  (set! x 10)

let*

```
(defn form-eval [exp env]
    (cond (self-evaluating? exp)  exp
          ;...
          (= (first exp) 'let*)  (eval-let* exp env))) ;<--here

(defn eval-let* [exp env]
  (let [eenv (extend-env env)
        [op vars body] exp]
    (doseq [[b e] vars]
      (swap! (first eenv) assoc b (form-eval e eenv)))
    (form-eval body eenv)))
```

- (let* ((var val) ...) body)

defmacro

```
(defn form-eval [exp env]
    (cond (self-evaluating? exp)  exp
          ;...
          (= (first exp) 'defmacro) (eval-defmacro exp env)));<-here

(defn eval-defmacro [exp env]
    (let [[a0 a1 a2] exp
          mbody (with-meta (form-eval a2 env) {:ismacro true})]
      (swap! (first env) assoc a1  mbody)
      "ok"))
```

- (defmacro binding (lambda (args) body))

macroexpand

```
(defn form-eval [exp env]
  (let [exp (macroexpand exp env)] ;<--here
    (cond (self-evaluating? exp)  exp
          (= (first exp) 'macroexpand) (macroexpand (second exp) env)))) ;<-- here

(defn macroexpand [exp env]
  (loop [exp exp]
    (if (is-macro-call exp env)
      (let [mac (env-get (first exp) env)]
        (recur (form-apply mac (rest exp))))
      exp)))
```

macroexpand (cont.)

```
(defn is-macro-call [exp env]
  (and (seq? exp)
       (symbol?  (first exp))
       (env-find (first exp)
                 env
                 #(:ismacro (meta (get @% (first exp))))
                 #(identity false))))
```

to apply

```
(defn form-eval [exp env]
  (let [exp (macroexpand exp env)]
    (cond (self-evaluating? exp)  exp
          (symbol? exp)           (env-get exp env)
          (= (first exp) 'quote)  (second exp)
          (= (first exp) 'if)     (eval-if exp env)
          (= (first exp) 'begin)  (eval-seq (rest exp) env)
          (= (first exp) 'lambda) (eval-lambda exp env)
          (= (first exp) 'define) (eval-define exp env)
          (= (first exp) 'set!)   (eval-assignment exp env)
          (= (first exp) 'let*)   (eval-let* exp env)
          (= (first exp) 'defmacro) (eval-defmacro exp env)
          (= (first exp) 'macroexpand) (macroexpand (second exp) env)
          :else (form-apply (form-eval (first exp) env)  ;<-- here
                            (map #(form-eval % env) (rest exp))))))
```

apply

```
(defn form-apply [proc args]
  (letfn [(primitive? [p] (= (first p) 'primitive))
          (apply-primitive [p a] (apply (second p) a)) ;<-- magic
          (procedure? [p] (= (first p) 'procedure)) ;<-- from lambda
          (proc-params [p] (second p))
          (proc-body [p] (nth p 2))
          (proc-env [p] (nth p 3))]
    (cond (primitive? proc) (apply-primitive proc args)
          (procedure? proc) (eval-seq (proc-body proc)
                                      (extend-env (proc-env proc)
                                                  (proc-params proc)
                                                  args)))))
```

Now you can write your own Lisp in Clojure

Try it

```
(def env (setup-env))

(form-eval
  '(define (fact x)
     (if (eq? x 1)
         1
         (* x (fact (- x 1))))) env) ;=> ok

(form-eval
  '(fact 6) env)) ;=> 720
```

Something else

- Add your built-in functions
- REPL

Question?