```
#lang racket
(define (mini-eval expr env)
  (display "  going to evaluate expression: ")
  (displayln expr)
  (cond
    ((number? expr) expr)
    ((symbol? expr) (lookup-variable-value expr env))

    ((add? expr) (eval-add expr env))
    ((subtract? expr) (eval-subtract expr env))
    ((multiply? expr) (eval-multiply expr env))
    ((exponent? expr) (eval-exponent expr env))

    ((definition? expr) (eval-definition expr env))
    ((ifzero? expr) (eval-ifzero expr env))
    ((lambda? expr) (eval-lambda expr env))
    ((call? expr) (eval-call expr env))

    (#t (error "kablooie: " expr))
    ))

(define (add? expr) (equal? 'add (car expr)))
(define (subtract? expr) (equal? 'sub (car expr)))
(define (multiply? expr) (equal? 'mul (car expr)))
(define (exponent? expr) (equal? 'exp (car expr)))
(define (definition? expr) (equal? 'define (car expr)))
(define (ifzero? expr) (equal? 'ifzero (car expr)))
(define (call? expr) (equal? 'call (car expr)))
(define (lambda? expr) (equal? 'lambda (car expr)))

; For eval-add, expr is always of the form: (add expr1 expr2)
;   Both sub-expressions must eval to numbers.
(define (eval-add expr env)
  (+ (mini-eval (cadr expr) env) (mini-eval (caddr expr) env)))

; For eval-subtract, expr is always of the form: (sub expr1 expr2)
;   Both sub-expressions must eval to numbers.
(define (eval-subtract expr env)
  (- (mini-eval (cadr expr) env) (mini-eval (caddr expr) env)))

; For eval-multiply, expr is always of the form: (mul expr1 expr2)
;   Both sub-expressions must eval to numbers.
(define (eval-multiply expr env)
  (* (mini-eval (cadr expr) env) (mini-eval (caddr expr) env)))

; For eval-exponent, expr is always of the form: (exp expr1 expr2)
;   Both sub-expressions must eval to numbers.
(define (eval-exponent expr env)
  (expt (mini-eval (cadr expr) env) (mini-eval (caddr expr) env)))

; For eval-definition, expr is always of the form: (define var expr1)
;   var must be a variable name (a symbol), and expr1 can eval to anything.
(define (eval-definition expr env)
  (hash-set! (car env) (cadr expr) (mini-eval (caddr expr) env)))

; For eval-ifzero, expr is always of the form: (ifzero expr1 expr2 expr3)
;   expr1 must evaluate to a number, and expr2 and expr3 can evaluate to anything.
(define (eval-ifzero expr env)
  (if (= 0 (mini-eval (cadr expr) env))
      (mini-eval (caddr expr) env)
      (mini-eval (cadddr expr) env)))

; For eval-lambda, expr is always of the form: (lambda argname body)
;   argname must be a variable name (a symbol), and body is any expression.
(define (eval-lambda expr env)
  (list 'closure (cadr expr) (caddr expr) env))

; For eval-call, expr is always of the form: (call expr1 expr2)
;   expr1 must eval to a closure, expr2 can evaluate to anything.
(define (eval-call expr env)
  (mini-apply (mini-eval (cadr expr) env) (mini-eval (caddr expr) env)))

(define (lookup-variable-value var env)
  (cond ((hash-has-key? (car env) var) (hash-ref (car env) var))
        ((null? env) (error "unbound variable" var))
        (#t (lookup-variable-value var (cdr env)))))
```

```
(define (mini-apply closure argval)
  (let ((new-frame (make-hash)))
    (hash-set! new-frame (cadr closure) argval)
    (let ((new-env (cons new-frame (cadddr closure))))
      (mini-eval (caddr closure) new-env))))

(define (run)
  (let ((global-env (list (make-hash))))
    (define (read-eval-print-loop)
      (display "mini-eval input: ")
      (let ((input (read)))
        (if (not (equal? input 'end))
            (let ((output (mini-eval input global-env)))
              (display "mini-eval output: ")
              (displayln output)
              (read-eval-print-loop))
            'done)))
    (read-eval-print-loop)))

; (define square (lambda p (mul p p)))
; (define fact (lambda n (ifzero n 1 (mul n (call fact (sub n 1))))))
; (define fib (lambda n (ifzero n 0 (ifzero (sub n 1) 1 (add (call fib (sub n 1)) (call fib (sub n 2)))))))
; (define make-adder (lambda s (lambda t (add s t))))
; (define add1 (call make-adder 1))
; (call add1 4)
```