

```
; Datatype for procedures. At first there is only one
; kind of procedure, but more kinds will be added later.
```

```
(define-datatype proc-val proc-val?
  [prim-proc
   (name symbol?)])

(define-datatype expression expression?
  [var-exp      ; variable references
   (id symbol?)]
  [lit-exp
   (datum (lambda (x)
              (ormap
               (lambda (pred) (pred x))
               (list number? vector? boolean?
                     symbol? string? pair? null?)))))]
  [app-exp      ; applications
   (rator expression?)
   (rands (list-of expression?))]
  [else (eopl:error 'expression "not an expression")]
  )
```

```
; datatypes and procedures for environments are in the
; code and in the slides
```

```
; The main interpreter cod:
```

```
(define eval-exp ;additional comments
  (lambda (exp) ; are on PPT slides
    (cases expression exp
      [lit-exp (datum) datum]
      [var-exp (id)
       (apply-env init-env)]
      [app-exp (rator rands)
       (let ([proc-value (eval-exp rator)]
             [args (eval-rands rands)])
         (apply-proc proc-value args))]
      [else (eopl:error 'eval-exp
                        "Bad abstract syntax: ~a"
                        exp)]))
```

```
; Evaluate list of operands, return list of results
```

```
(define eval-rands (lambda (rands)
                     (map eval-exp rands)))
```

```
; Apply a procedure to its arguments.
```

```
(define apply-proc
  (lambda (proc-value args)
    (cases proc-val proc-value
      [prim-proc (op) (apply-prim-proc op args)]
      [else (error 'apply-proc
                    "Attempt to apply bad procedure: ~s"
                    proc-value)])))
```

```
(define apply-prim-proc
  (lambda (prim-proc args)
    (case prim-proc
      [(+) (+ (1st args) (2nd args))]
      [(-) (- (1st args) (2nd args))]
      [(*) (* (1st args) (2nd args))]
      [(add1) (+ (1st args) 1)]
      [(sub1) (- (1st args) 1)]
      [(cons) (cons (1st args) (2nd args))]
      [(=) (= (1st args) (2nd args))]
      [else (error 'apply-prim-proc
                    "Bad primitive procedure name: ~s"
                    prim-op)])))
```

```
; Set up the initial environment.
```

```
(define *prim-proc-names* '(+ - * add1 sub1 cons =))
```

```
(define init-env ; For now, our initial environment
  (extend-env ; only contains procedure names.
    *prim-proc-names* ; An environment
    (map prim-proc ; associates values (not
      *prim-proc-names*) ; expressions) with
    (empty-env)) ; variables.
```

```
(define rep ; "read-eval-print" loop.
  (lambda ()
    (display "--> ")
    (let ([answer (top-level-eval (parse-exp (read)))]
          (eopl:pretty-print answer)
          (newline)
          (rep)))) ; tail-recursive, so stack doesn't grow.
```