3.5 Procedures

- Recall procedures (functions) in Scheme:

```
(let ((f (lambda(y z) (+ y (- z 5))))
      (f 2 28))
```

- We would like something similar in our toy language:

```
let f = proc (y, z) +(y, -(z, 5))
    in (f 2 28)
```

3.5 Procedures

Syntax:

<expression> ::= **proc (**{<identifier>}*(,) **)**

<expression>

```
proc-exp (ids body)
```

::= **(**<expresion> {<expression>}***)**

```
app-exp (rator rands)
```

3.5 Procedures: Step 1

Add to grammar:

```
define grammar-3-5
 '((program (expression) a-program)
   (expression (number) lit-exp)
   . . .
   (expression (
          "proc"
          "("
           (separated-list identifier ",")
          ")"
          expression) ; body
      proc-exp)
   (expression (
         "("
         expression          ; 'rator
         (arbno expression) ; 'rands
         ")")
      app-exp)
```

3.5 Procedures: Step 2

Extend datatype definition for expression (not in book):

```
define-datatype expression expression?
  ...
  (proc-exp
     (ids  (list-of symbol?))
     (body  expression?))
  (app-exp
     (rator expression?)
     (rands (list-of expression?))
```

3.5 Procedures: Step 3

- Need to add some code to **`eval-expression`**
- But first we must understand procedure semantics:

```
let x = 5
  in let f = proc(y, z) +(y, -(z,x))
         x = 28
       in (f 2 x)
```

- Which value of `x` (5 or 28) is used in the `let f...` line?
- Which value of `x` (5 or 28) is used in the last line?
- Why?

3.5 Procedures

```
let x = 5
   in let f = proc(y, z) +(y, -(z,x))
            x = 28
         in (f 2 x)
```

- The value `x = 28` is used in the last line.

- The value `x = 5` is used in the second-to-top line, because that is the most recent definition "seen" by the *definition* of `f`.

- Again, we say that `x` is *free* in `f`.

3.5 Procedures

- Together, a procedure and its free variables form a *closure*: a "package" containing the proc and free vars:

```
(define-datatype procval procval?
   (closure
      (ids (list-of symbol?))
      (body expression?)
      (env environment?)))  ; free vars
```

- We use this code as a constructor in **`eval-expression`**:

3.5 Procedures

```
(define eval-expression
   (lambda (exp env)
      (cases expression exp
      ...
         (proc-exp (ids body)
            (closure ids body env))))
```

- Now we can handle a procedure *application*:

3.5 Procedures

```
(define eval-expression
    (lambda (exp env)
        (cases expression exp
            ...
            (app-exp (rator rands)
                (let ((proc (eval-expression rator env))
                      (args (eval-rands rands env)))
                    (if (procval? proc)
                        (apply-procval proc args)
                        (eopl:error 'eval-expression
                            “Applied non-procedure ~s”
                            proc))))
```

3.5 Procedures

```
(define apply-procval
  (lambda (proc args)
    (cases procval proc ;only one case
      (closure (ids body env)
        (eval-expression body
          (extend-env ids args env))))))
```

`apply-procval`: Example

```
let x = 3
  in let f = proc(y) +(x, y)
       in (f 2)

ids:   ‘(y)
body:  <+(x,y)>
env:   x: 3

(eval-expression <+(x,y)>
   (extend-env ‘(y) ‘(2) ‘((x 3))))

(eval-expression <+(x,y)> ‘((y 2) (x 3)))
```