3.6 Interpreter: Recursion

- Recall Scheme's **`letrec`** (recursive **`let`**):

```
> (letrec ((fact (lambda (x)
                   (if (zero? x)
                   1
                   (* x (fact (- x 1))))))
    (fact 6)) ; body of the letrec
720
```

- Question: Why **`letrec`** and not **`let`** ?

3.6 Recursion

- Implement **`letrec`** for procedures in our toy language:

```
--> letrec fact(x)=
                  if zero?(x) then 1
                  else *(x, (fact sub1(x)))
    in (fact 6)
720
-->
```

- We'll use **`letrec`** only for procedures....

3.6 Recursion

- Syntax:

<expression>

::= **letrec**

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

in <expression>

```
letrec-exp
 (proc-names idss bodies
  letrec-body)
```

- Question: Why **idss** and not **ids** ?

3.6 Recursion

- Add some code to **`eval-expression`**:

```
(define eval-expression
  (lambda (exp env)
    (cases expression exp
        ...
      (letrec-exp (proc-names idss bodies letrec-body)
         (eval-expression letrec-body
          (extend-env-recursively
            proc-names idss bodies env)))
      ...)))
```

- So how to write **`extend-env-recursively`** ?

3.6 Recursion

First augment environment type to include recursive procs:

```
(define-datatype environment environment?
    (empty-env-record)
    (extended-env-record
      (syms (list-of symbol?))
      (vals (list-of scheme-value?))
      (env  environment?))
 (recursively-extended-env-record
     (proc-names (list-of symbol?))
     (idss (list-of (list-of symbol?)))
     (bodies (list-of expression?))
     (env environment?)))
```

3.6 Recursion

- Now write **`extend-env-recursively`** as a constructor:

```
(define extend-env-recursively
  (lambda (proc-names idss bodies env)
    (recursively-extended-env-record
      proc-names idss bodies env)))
```

- Recall that we look up symbols using **`apply-env:`**

```
(define eval-expression
 (lambda (exp env)
   (cases expression exp
       (var-exp (id) (apply-env env id)
```

3.6 Recursion

Current version:

```
(define apply-env
  (lambda (env sym)
    (cases environment env
      (empty-env-record ()
        (eopl:error ...))
      (extended-env-record (syms vals old-env)
        (let ((pos (list-find-pos sym syms)))
          (if (number? pos) ; found
              (list-ref vals pos)
              (apply-env old-env sym)))))))
```

3.6 Recursion

```
(define apply-env
  (lambda (env sym)
    (cases environment env
      ...
      (recursively-extended-env-record
            (proc-names idss bodies old-env)
        (let ((pos (list-find-pos sym proc-names)))
          (if (number? pos); found
              (closure (list-ref idss pos)
                       (list-ref bodies pos)
                       env) ; circular!
              (apply-env old-env sym)))))));notfound
```

3.6 Recursion

- For our **fact** example:

```
proc-names = '(fact)

idss = '((x))

bodies = '( <<if zero?(x) then 1

             else *(x, fact sub1(x))>> )

env = '( (fact)

         (<closure: fact, x, <<if...) )
```

3.6 Recursion