$\boxed{\text{Contexts and Closures}}$

- "**`eval`**" will **`subst`** (for free var's) immediately
- Correct ... but not very efficient:

  One pass to find free vars, another for subst.

  ... renaming rule ($\mapsto^6$)...

  Why subst for **`else`**, if take **`then`** branch?

Idea: Don't do substitutions "on the spot"

... delay it, until needed!

$\approx$ Can SUSPEND **`subst`** temporarily

- Just need to remember "context"

  and carry it with expression

- "Lazy Evaluation"

$\boxed{\text{Example of “Context”}}$

Eg: `((`λ`x | (x + 4)) 2)`

can be interpreted as

`eval` `x+4` in context: `x` is 2

written “{ `x` $\rightsquigarrow$ 2 }”

Note: Don't need to subst `x` by 2
when evaluating function application

...just need to REMEMBER this context

- **context** $\equiv$ dictionary of name bindings,
  used to record these substitutions
  and to “look-up” binding values.

- **closure** $\equiv$ everything required to eval λ-expr
  $+$ λ-expression (body and parameters)
  $+$ context used for (free) var's in body
  $\boxed{\text{(shown as box)}}$

Use of Context

Eg: `let x = 3 in x + 1`

$\equiv$ `((`λ`x | x + 1) 3)`

Eval body "`(x + 1)`" in context $\{$`x` $\leadsto$ `3`$\}$

... `eval[ (x + 1),` $\{$`x` $\leadsto$ `3`$\}$ `]`

Eg: `(`λ`x | ((`λ`y|y) x+1)) 2`

$\mapsto$ `(`λ`y|y) 2+1` ie, subst `x` (in `x+1`) by 2

$\mapsto$ `(`λ`y|y) 3`

$\mapsto$ `3` ie, subst `y` (in `y`) by 3

- During process...

  $\mapsto$ `(`λ`y|y) x+1` where $\{$`x` $\leadsto$ `2`$\}$

  or later...

  $\mapsto$ `y` where $\{$`y` $\leadsto$ `3, x` $\leadsto$ `2`$\}$

- Need to modify `eval`, `apply` to take "context" arg

  Each time processing *application*,

  eval its $\left\{ \begin{array}{l} \text{function} \\ \text{args} \end{array} \right\}$ in current context

$\boxed{\textbf{Evaluation, wrt Context}}$

- To eval *identifier wrt context*,
  use binding of identifier in context

Eg: `eval[x,` $\{$`x` $\leadsto$ `1`$\}$`]  =  1`

- To eval *function definition wrt context*
  return closure containing: function, context

Eg: `eval[(`λ`z | z + x),` $\{$`x` $\leadsto$ `1`$\}$`]`

$$\Longrightarrow \quad \boxed{(\lambda z \mid z + x),\ \{x \leadsto 1\}}$$

- To eval an *application wrt context*,
  1. eval function, args in current context
  2. bind formal vars to evaluated args
  3. extend function's context by
     appending these bindings to it
  4. eval body of function in extended context

Eval of Application (Example)

`let x = 1 in let y = (`λ`z | z + x) in y(3)`

$\equiv$

`eval[ ((`λ`x | (`λ`y | (y 3)) (`λ`z | z + x)) 1), {} ]`

(1a) eval fn:

`eval[(`λ`x | (`λ`y | (y 3)) (`λ`z | z + x)), {}]`

$\Longrightarrow$ `(`λ`x | (`λ`y | (y 3)) (`λ`z | z + x)), {}`

body = `(`λ`y | (y 3)) (`λ`z | z + x)`

(1b) eval arg:

`eval[1, {}]` $\Longrightarrow$ `1`

(2,3) Extend earlier context {} by binding arg `x` to value `1`

New context: $\{\mathtt{x} \rightsquigarrow 1\} \cup \{\}$

---

4. Now recur:
eval body `(`λ`y | (y 3)) (`λ`z | z + x)`
in context $\{\mathtt{x} \rightsquigarrow 1\}$

`eval[ (`λ`y | (y 3)) (`λ`z | z + x), {x` $\rightsquigarrow$ `1} ]`

NOTE: did NOT substitute `x` by `1` in body!

. . . continue . . .

$\boxed{\text{Eval Application - Con't}}$

eval[$(\lambda$ y | (y 3)) $(\lambda$ z | z + x), $\{$x $\rightsquigarrow$ 1$\}$]

Eval fn:

eval[$(\lambda$ y | (y 3)), $\{$x $\rightsquigarrow$ 1$\}$] $\Longrightarrow$

$\boxed{(\lambda\, \mathtt{y} \mid (\mathtt{y}\ 3)),\ \{\mathtt{x} \rightsquigarrow 1\}}$

body = (y 3)

Eval arg:

eval[$(\lambda$ z | z + x), $\{$x $\rightsquigarrow$ 1$\}$]

$\Longrightarrow$ $\boxed{(\lambda\, \mathtt{z} \mid \mathtt{z} + \mathtt{x}),\ \{\mathtt{x} \rightsquigarrow 1\}}$

New context:

$\{\mathtt{y} \rightsquigarrow \boxed{(\lambda\, \mathtt{z} \mid \mathtt{z} + \mathtt{x}),\ \{\mathtt{x} \rightsquigarrow 1\}},\ \mathtt{x} \rightsquigarrow 1\}$

NOTE: y is bound to CLOSURE!

Now eval body (y 3)

in context $\{\mathtt{y} \rightsquigarrow \boxed{(\lambda\, \mathtt{z} \mid \mathtt{z} + \mathtt{x}),\ \{\mathtt{x} \rightsquigarrow 1\}},\ \mathtt{x} \rightsquigarrow 1\}$

eval[(y 3), $\{\mathtt{y} \rightsquigarrow \boxed{(\lambda\, \mathtt{z} \mid \mathtt{z} + \mathtt{x}),\ \{\mathtt{x} \rightsquigarrow 1\}},\ \mathtt{x} \rightsquigarrow 1\}$]

Eval fn:

eval[y, $\{\mathtt{y} \rightsquigarrow \boxed{(\lambda\, \mathtt{z} \mid \mathtt{z} + \mathtt{x}),\ \{\mathtt{x} \rightsquigarrow 1\}},\ \mathtt{x} \rightsquigarrow 1\}$]

$\Longrightarrow$ $\boxed{(\lambda\, \mathtt{z} \mid \mathtt{z} + \mathtt{x}),\ \{\mathtt{x} \rightsquigarrow 1\}}$

body = z + x

Eval arg:

$\texttt{eval[3, }\{\texttt{y} \leadsto \boxed{\texttt{(}\lambda\,\texttt{z | z + x), }\{\texttt{x} \leadsto \texttt{1}\}}\texttt{, x} \leadsto \texttt{1}\}\texttt{]}$

$\Longrightarrow \quad \texttt{3}$

New context $=$ *earlier context* $\{\texttt{x} \leadsto \texttt{1}\}$

$+$ *new context* $\{\texttt{z} \leadsto \texttt{3}\}$

$= \{\texttt{z} \leadsto \texttt{3, x} \leadsto \texttt{1}\}$

Now eval body `(z + x)`

in context $\{\texttt{z} \leadsto \texttt{3, x} \leadsto \texttt{1}\}$

$\Longrightarrow \texttt{eval[z + x, }\{\texttt{z} \leadsto \texttt{3, x} \leadsto \texttt{1}\}\texttt{]} \qquad \Longrightarrow \texttt{4}$

Overriding Binding

New bindings can overide old:

`let x = 1 in let y = (λz | z + x) in let x = 5 in y(x)`

eval[((λx | (λy | (λx | (y x)) 5) (λz | z + x)) 1), {}]

Eval fn:

eval[(λx | (λy | (λx | (y x)) 5) (λz | z + x)), {}]

$\Longrightarrow$ $\boxed{\texttt{(λx | (λy | (λx | (y x)) 5) (λz | z + x)), \{\}}}$

body = (λy | (λx | (y x)) 5) (λz | z + x)

Eval arg:

eval[1, {}] $\Longrightarrow$ 1

New context: {x $\leadsto$ 1}

---

eval[(λy | (λx | (y x)) 5) (λz | z + x), {x $\leadsto$ 1}]

Eval fn:

eval[(λy | (λx | (y x)) 5), {x $\leadsto$ 1}]

$\Longrightarrow$ $\boxed{\texttt{(λy | (λx | (y x)) 5), \{x} \leadsto \texttt{1\}}}$

body = ((λx | (y x)) 5)

Eval arg:

eval[(λz | z + x), {x $\leadsto$ 1}]

$\Longrightarrow$ $\boxed{\texttt{(λz | z + x), \{x} \leadsto \texttt{1\}}}$

New context: $\{y \rightsquigarrow \boxed{(\lambda z \mid z + x),\ \{x \rightsquigarrow 1\}},\ x \rightsquigarrow 1\}$

$\texttt{eval}[((\lambda x \mid (y\ x))\ 5),$
$\quad \{y \rightsquigarrow \boxed{(\lambda z \mid z + x),\ \{x \rightsquigarrow 1\}},\ x \rightsquigarrow 1\}]$

Eval fn:

$\texttt{eval}[(\lambda x \mid (y\ x)),$
$\quad \{y \rightsquigarrow \boxed{(\lambda z \mid z + x),\ \{x \rightsquigarrow 1\}},\ x \rightsquigarrow 1\}] \Longrightarrow$

$\boxed{(\lambda x \mid (y\ x)),\ \{y \rightsquigarrow \boxed{(\lambda z \mid z + x),\ \{x \rightsquigarrow 1\}},\ x \rightsquigarrow 1\}}$

body $= (y\ x)$

Eval arg: $\texttt{eval}[5,\ \{\ldots\}] \Longrightarrow 5$

New context:

$\{x \rightsquigarrow 5,\ y \rightsquigarrow \boxed{(\lambda z \mid z + x),\ \{x \rightsquigarrow 1\}},\ x \rightsquigarrow 1\}$

Note order of OrderedList!

$\texttt{eval}[(y\ x),$
$\quad \{x \rightsquigarrow 5,\ y \rightsquigarrow \boxed{(\lambda z \mid z + x),\ \{x \rightsquigarrow 1\}},\ x \rightsquigarrow 1\}]$

Eval fn:

$\texttt{eval}[y,\ \{x \rightsquigarrow 5,\ y \rightsquigarrow \boxed{(\lambda z \mid z + x),\ \{x \rightsquigarrow 1\}},\ x \rightsquigarrow 1\}]$

$\Longrightarrow \boxed{(\lambda z \mid z + x),\ \{x \rightsquigarrow 1\}}$

body $= z + x$

Eval arg: `eval[x,`

$\{$`x` $\leadsto$ `5, y` $\leadsto$ $\boxed{\texttt{(}\lambda\texttt{z | z + x), } \{\texttt{x} \leadsto \texttt{1}\}}$`, x` $\leadsto$ `1`$\}$`]`

$\Longrightarrow$ `5`

Note: Returns FIRST value!

New context: $\{$`z` $\leadsto$ `5, x` $\leadsto$ `1`$\}$

`eval[z + x,` $\{$`z` $\leadsto$ `5, x` $\leadsto$ `1`$\}$`]`

$\Longrightarrow$ `eval[5 + 1]` $\Longrightarrow$ `6`

Implementing Contexts

- Implement **context dictionaries** using
  2 parallel lists
  list#1 for names
  list#2 for values

Eg: Context for

```
let x = 1 in let y = (λz | z + x) and w = 2 in
   let x = x + w in y(x)
```

represented by name and value lists:

```
Name:   ((x)   (y                        w)   (x))
Value:  ((1)   ((lambda (z) (+ z x))    2)   ((+ x 1)))
```

Utility Functions

- `assoc[x, n, v]` takes
  atom `x`, name list `n`, value list `v`
  returns value `x`
  or “undefined” if `x` is not found

  Here `n`, `v` are list of LISTS
  one per “function call”

```
assoc[x, n, v] =
   if member[x, car[n]] then locate[x, car[n], car[v]]
   else assoc[x, cdr[n], cdr[v]]

locate[x, k, m] =
   if eq[x, car[k]] then car[m]
   else locate[x, cdr[k], cdr[m]]

member[x, k] =
   if eq[k, NIL] then false
   else if eq[x, car[k]] then true
   else member[x, cdr[k]]
```

Example . . .

```
assoc[w, ((x) (y w) (x)),
      ((1) ((lambda (z) (+ z x)) 2) ((+ x 1)))]
member[w, car[((x) (y w) (x))]] = member[w, (x)] = false
```

SO. . .

```
= assoc[w, cdr[((x) (y w) (x))],
        cdr[((1) ((lambda (z) (+ z x)) 2) ((+ x 1)))]]
= assoc[w, ((y w) (x)),
        (((lambda (z) (+ z x)) 2) ((+ x 1)))]

member[w, car[((y w) (x))]] = member[w, (y w)] = true
```

SO. . .

```
= locate[w, car[((y w) (x))],
        car[(((lambda (z) (+ z x)) 2) ((+ x 1)))]]

= locate[w, (y w), ((lambda (z) (+ z x))) ((+ x 1) 2)]

eq[w, car[(y w)]] = eq[w, y] = false
```

SO. . .

```
= locate[w, cdr[(y w)],
        cdr[((lambda (z) (+ z x))) ((+ x 1) 2)]]
= locate[w, (w), (2)]

eq[w, car[(w)]] = eq[w, w] = true
```

SO. . .

```
= car[(2)] = 2
```

Interpreter Function (Final)

- `eval[e, n, v]` returns
  result of evaluating expression `e`,
  using context:

  name list `n`, value list `v`

  . . . for *any* extended lambda expression

- Every expression can be built from well-formed expressions

Notation: `e, e1, e2, ...` denote well formed expressions

`x, x1, x2, ...` denote atoms used as variables

`n, n1, n2, ...` denote name lists

`v, v1, v2, ...` denote value lists

`a, b, s, ...` denote arbitrary S-exprs

Well-Formed Expressions

Need to specify result of applying `eval`
to each of these
18 well-formed expressions:

`x`	`a variable`
`(quote s)`	`a constant`
`(+ e1 e2)`	
`(- e1 e2)`	
`(* e1 e2)`	`arithmetic expressions`
`(/ e1 e2)`	
`(rem e1 e2)`	
`(eq e1 e2)`	`relational expressions`
`(leq e1 e2)`	
`(car e)`	`structural expressions`
`(cdr e)`	
`(cons e1 e2)`	
`(atom e)`	
`(if e1 e2 e3)`	`conditional form`
`(lambda (x1... xk) e)`	`lambda expression`
`(e e1... ek)`	`function call`
`(let (x1.e1)...(xk.ek) e)`	`simple block`
`(letrec (x1.e1) ... (xk.ek) e)`	`recursive block`

Evaluation #1

- **variable**

  `eval[x, n, v] = assoc[x, n, v]`

- **constant**

  `eval[(quote s), n, v] = s`

- Eval of

  arithmetic, relational, structural expr's

  . . . defined in terms of `eval` of arg(s) in expr'n

```
eval[(car e), n, v] = car[ eval[e, n, v] ]
eval[(cdr e), n, v] = cdr[ eval[e, n, v] ]
eval[(cons e1 e2), n, v] = cons[eval[e1, n, v], eval[e2, n, v]]

eval[(atom e), n, v] = atom[eval[e, n, v]]
eval[(eq e1 e2), n, v] = eq[eval[e1, n, v], eval[e2, n, v]]
eval[(leq e1 e2), n, v] = leq[eval[e1, n, v], eval[e2, n, v]]

eval[(+ e1 e2), n, v] = eval[e1, n, v] + eval[e2, n, v]
...
```

Evaluation #2: If

- `Eval` of **conditional expressions**

  Note: cannot initially evaluate all arguments of conditional expression
  before evaluating conditional expression itself

  . . . Normal Order . . .

  . . . only first arg always evaluated,

  then either 2nd or 3rd arg is eval'ed

  depending on value of first arg.

```
eval[(if e1 e2 e3), n, v] =
   if eval[e1, n, v]
      then eval[e2, n, v]
      else eval[e3, n, v]
```

Evaluation #3: LAMBDA

- `Eval` of **lambda expressions**

  λ-expression evals to **closure**, containing

  body of lambda expression

  variable list

  context in which body must be eval'ed

  when function is eventually applied...

  Parts of closure `C`:

  `parms[C], body[C], names[C], values[C]`

- Store information in list of form:

```
eval[(lambda y e), n, v]
   =  cons[cons[y, e], cons[n, v]]
   =  [ [y .  e] .  [n .  v] ]
```

Eg:
```
eval[(lambda (z) (+ z x)), ((x)), ((1))] =
cons[cons[(z), (+ z x)], cons[((x)), ((1))]] =
cons[((z) + z x), (((x)) (1)) = (((z) + z x) ((x)) (1))
```

Evaluation #4

- `eval` for **function calls:**

```
eval[(e e1 ...  ek), n, v] =
 eval[body[c],  cons[parms[c],  names[c]],
                cons[z,         values[c]] ]
```

where `c = eval[e, n, v],`

```
       z = evlis[(e1 ...  ek), n, v]
```

- Using `evlis`

  to eval arg list, one arg at a time:

```
evlis[k, n, v] =
  if k = nil then nil
  else cons[ eval[car[k], n, v], evlis[cdr[k], n, v] ]
```

Examples of `evlis`

Eg1: `evlis[((+ q 4) (+ q x)), ((q) (x)), ((3) (1))]`

```
   As  ((+ q 4) (+ q x)) ≠ nil, ...
= cons[ eval[car[((+ q 4) (+ q x))],
              ((q) (x)), ((3) (1))],
   evlis[ cdr[((+ q 4) (+ q x))], ((q) (x)),
              ((3) (1))]]
= cons[eval[(+ q 4), ((q) (x)), ((3) (1))],
          evlis[((+ q x)), ((q) (x)), ((3) (1))]]

   As ((+ q x)) ≠ nil, ...
= cons[ 7, cons[ eval[car[((+ q x))],
                ((q) (x)), ((3) (1))],
     evlis[cdr[((+ q x))], ((q) (x)), ((3) (1))]]]
= cons[7, cons[eval[(+ q x), ((q) (x)), ((3) (1))],
       evlis[nil, ((q) (x)), ((3) (1))]]]
= cons[7, cons[4, nil]] = (7 4)
```

Eg2: `evlis[((lambda (z) (+ z x))), ((x)), ((1))]`

```
   As ((lambda (z) (+ z x))) ≠ nil, ...
= cons[ eval[ car[((lambda (z) (+ z x)))],
                ((x)), ((1))],
   evlis[cdr[((lambda (z) (+ z x)))], ((x)), ((1))]]
```

```
= cons[ eval[ (lambda (z) (+ z x))], ((x)), ((1))],
              evlis[nil, ((x)), ((1))]]
= cons[(((z) + z x) ((x)) (1)), nil]
= ((((z) + z x) ((x)) (1)))
```

Example of eval Function Call

Eg: $\texttt{eval[((}\lambda\texttt{x | (}\lambda\texttt{y | (}\lambda\texttt{x | (y x)) 5) (}\lambda\texttt{z | z + x)) 1), }\{\}\texttt{]}$

```
eval[((lambda (x) (lambda (y) (lambda (x) (y x)) 5)
       (lambda (z) z + x)) 1), nil, nil]

e = (lambda (x) (lambda (y) (lambda (x) (y x)) 5)
       (lambda (z) z + x))
e1 = 1, n = nil, v = nil

c = eval[(lambda (x) (lambda (y) (lambda (x) (y x)) 5)
       (lambda (z) z + x)), nil, nil]

body[c] = (lambda (y) (lambda (x) (y x)) 5)
       (lambda (z) z + x)

parms[c] = (x), names[c] = nil, values[c] = nil

z = evlis[(1), nil, nil] = (1)

 ==> eval[ (lambda (y) (lambda (x) (y x)) 5)
       (lambda (z) (+ z x)),
   cons[(x), nil], cons[(1), nil]]
= eval[(lambda (y) (lambda (x) (y x)) 5) (lambda (z) (+ z x)),
       ((x)), ((1)) ]

e = (lambda (y) (lambda (x) (y x)) 5)

e1 = (lambda (z) z + x),  n = ((x)),  v = ((1))

c = eval[(lambda (y) (lambda (x) (y x)) 5), ((x)), ((1))]

body[c] = (lambda (x) (y x)) 5)
```

```
parms[c] = (y), names[c] = ((x)), values[c] = ((1))

evlis[((lambda (z) (+ z x))), ((x)), ((1))] = c1
```

where `body[c1] = (+ z x), parms[c1] = (z),`

```
   names[c1] = ((x)), values[c1] = ((1))

 ⟹ eval[(lambda (x) (y x)) 5), cons[(y), ((x))],
   cons[((((z) + z x) ((x)) (1))), ((1))]]
= eval[(lambda (x) (y x)) 5), ((y) (x))],
     (((((z) + z x) ((x)) (1))) (1))]]

e = (lambda (x) (y x)), e1 = 5, n = ((y) (x)),
   v = (((((z) + z x) ((x)) (1))) (1))
   c = eval[(lambda (x) (y x)), ((y) (x)),
      (((((z) + z x) ((x)) (1))) (1))]

body[c] = (y x)

parms[c] = (x), names[c] = ((y) (x)),
   values[c] = (((((z) + z x) ((x)) (1))) (1))

evlis[(5), ((y) (x)), (((((z) + z x) ((x)) (1))) (1))]
= (5)

 ⟹ eval[(y x), cons[(x), ((y) (x))],
     cons[(5), (((((z) + z x) ((x)) (1))) (1))]]

= eval[(y x), ((x) (y) (x)), ((5) ((((z) + z x) ((x))
(1))) (1))]

e = y, e1 = x, n = ((x) (y) (x)),
   v = ((5) ((((z) + z x) ((x)) (1))) (1))
   c = eval[y, ((x) (y) (x)), ((5) ((((z) + z x) ((x))
```

```
(1))) (1))]
     = (((z) + z x) ((x)) (1))

body[c] = (+ z x), parms[c] = (z), names[c] = ((x)),

values[c] = ((1))

evlis[(x), ((x) (y) (x)), ((5) ((((z) + z x) ((x)) (1)))
(1))] = (5)

 ==> eval[(+z x), cons[(z), ((x))], cons[(5), ((1))]]

= eval[(+z x), ((z) (x))], ((5) (1))]

==> eval[z, ((z) (x)), ((5) (1))] + eval[x, ((z) (x)),

((5) (1))]

==> 5 + 1 = 6
```

Evaluation of let

Recall: `let` $\approx$ function application

```
let x1 = s1 and x2 = s2 in s3
```

$$\equiv$$

```
(λ x1 x2 | s3) s1 s2
```

In gen'l,

```
(let (x1.e1) ...  (xk.ek) e) =

 ((lambda (x1 ...  xk) e) e1 ...ek)
```

```
eval[(let (x1.e1) ...  (xk.ek) e), n, v]
= eval[((lambda (x1 ...  xk) e) e1 ...ek), n, v]
= eval[body[c], cons[parms[c], names[c]],
               cons[z, values[c]] ]
```
where `c = eval[(lambda (x1 ...  xk) e), n, v]`
`z = evlis[(e1 ...  ek), n, v]`

Eg: `eval[(let (x.1) (let (y.(lambda (z) (+ z x)))`
`(let (x.5) (y x)))), nil, nil]`

$\Longrightarrow$

```
eval[(let (y.(lambda (z) (+ z x))) (let (x.5) (y x))),
   cons[(x), nil], cons[z, nil]]
```

where `evlis[(1), nil, nil] = (1)`

```
= eval[(let (y.(lambda (z) (+ z x))) (let (x.5) (y x))),
((x)), ((1))]
```

$\Longrightarrow$ `eval[(let (x.5) (y x)), cons[(y), ((x))], cons[z, ((1))]]`

where z = evlis[((lambda (z) (+ z x))), ((x)), ((1))] =

((((z) + z x) ((x)) (1))) (by a previous example)

= eval[(let (x.5) (y x)), ((y) (x)), (((((z) + z x) ((x)) (1))) (1))]

$\Longrightarrow$ eval[(y x), cons[(x), ((y) (x))],

cons[z, (((((z) + z x) ((x)) (1))) (1))]]

where z = evlis[(5), ((y) (x)), (((((z) + z x) ((x)) (1))) (1))] = (5)

= eval[(y x), ((x) (y) (x)), ((5) ((((z) + z x) ((x)) (1))) (1))]

$\Longrightarrow$ 6 (by a previous example)

Evaluation of letrec

- `Eval` of **letrec expressions**

  Problem in defining recursive block:

  `(letrec (x1 .  e1) ...  (xk .  ek) E )`

  Expressions `e1 ...  ek` must be evaluated in environment that includes

  original bindings of `x1 ...  xk`

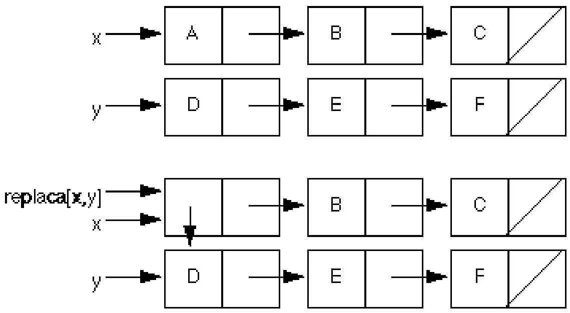
  Also: when recur, need to bring up NEW copy of function!

- To solve this problem, define pseudo-function, `replaca`

  `rplaca[x, y] = cons[y, cdr[x]]`

  “replace the car of x by y”

  + side effect that `(car x)` is now `y`

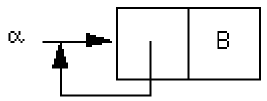


- $+$ special "PENDING" value: Ω.

$$\alpha = \text{rplaca[cons}[\Omega, \text{B], cons}[\Omega, \text{B]]}$$
$$= \quad \text{cons[cons}[\Omega, \text{B], B]}$$

with Ω re-bound to cons[Ω, B]

so car[α] $=$ cons[Ω, B] and cdr[α] $=$ B,
but Ω is re-bound to cons[Ω, B]:



- Use circular structure to define `letrec`

Recall def'n of `let`:

```
eval[(let (x1.e1) ...  (xk.ek) e), n, v]
= eval[e, cons[(x1 ...  xk), n], cons[z, v]]
```

where `z = evlis[(e1 ...  ek), n, v]`

```
  = eval[e, n', cons[z, cdr[v']]]
```

where `z = evlis[(e1 ...  ek), n, v]`

```
   n' = cons[(x1 ...  xk), n]
   v' = cons[Ω, v]
```

- Define `letrec` using `n'`, `v'` when evaluating `z`:

  `eval[(letrec (x1.e1) ... (xk.ek) e), n, v]`

  `= eval[e, n', replaca[v', z]]`

  where `z = evlis[(e1 ... ek), n', v']`
  `n' = cons[(x1 ... xk), n]`
  `v' = cons[`Ω`, v]`

- Similar to def'n of `let`, but
  before evaluating block, replace `car[v']`
  by `z` (evaluated definition list)

Eg: `eval[(letrec (fact lambda (x) (if (eq x 0) 1`
`(* x (fact (- x 1))))) (fact q)), ((q)), ((2))]`
$\Longrightarrow$ `eval[(fact q), n', replaca[v', z]]`
where `z = evlis[((lambda (x) B)), n', v']`,
`n' = cons[(fact), ((q))] = ((fact) (q))`,
`v' = cons[`Ω`, ((2))] = (`Ω`(2))`

`B = (if (eq x 0) 1 (* x (fact (- x 1))))`
`z = cons[C, nil] = (C)` where `parms[C] = (x)`, `body[C]`
`= B`,
`names[C] = ((fact) (q))`, `values[C] = (`Ω`(2))`

$\Longrightarrow$ `eval[(fact q), ((fact) (q)), replaca[(Ω(2)), (C)]]`

`= eval[(fact q), ((fact) (q)), cons[(C), cdr[( (2))]]]`

where Ω is rebound to `(C)` so
`values[C] = ((C) (2))`

`= eval[(fact q), ((fact) (q)), ((C) (2))]`

$\Longrightarrow$ `eval[body[c], cons[parms[c], names[c]], cons[z, values[c]]]`

where $\hat{c}$ = eval[fact, ((fact) (q)), ((C) (2))] = C
`z = evlis[(q), ((fact) (q)), ((C) (2))] = (2)`

`= eval[B, cons[(x), ((fact) (q))], cons[(2), ((C) (2))]]`

`= eval[B, ((x) (fact) (q)), ((2) (C) (2))]`

`= eval[(if (eq x 0) 1 (* x (fact (- x 1)))), ((x) (fact) (q)), ((2) (C) (2))]`

`= eval[(* x (fact (- x 1))), ((x) (fact) (q)), ((2) (C) (2))]`

`= eval[x, ((x) (fact) (q)), ((2) (C) (2))] *`

`eval[(fact (- x 1)), ((x) (fact) (q)), ((2) (C) (2))]`

= 2 * eval[(fact (- x 1)), ((x) (fact) (q)), ((2) (C) (2))]

Now eval[(fact (- x 1)), ((x) (fact) (q)), ((2) (C) (2))]

$\Longrightarrow$ eval[body[c], cons[parms[c], names[c]], cons[z, values[c]]]

where c = eval[fact, ((x) (fact) (q)), ((2) (C) (2))] = C

and z = evlis[(- x 1), ((x) (fact) (q)), ((2) (C) (2))] = (1)

= eval[B, cons[(x), ((fact) (q))], cons[(1), ((C) (2))]]

= eval[B, ((x) (fact) (q)), ((1) (C) (2))]

= eval[(if (eq x 0) 1 (* x (fact (- x 1)))), ((x) (fact) (q)), ((1) (C) (2))]

= eval[(* x (fact (- x 1))), ((x) (fact) (q)), ((1) (C) (2))]

= eval[x, ((x) (fact) (q)), ((1) (C) (2))] *

eval[(fact (- x 1)), ((x) (fact) (q)), ((1) (C) (2))]

= 1 * eval[(fact (- x 1)), ((x) (fact) (q)), ((1) (C) (2))]

Now `eval[(fact (- x 1)), ((x) (fact) (q)), ((1) (C) (2))]`

$\Longrightarrow$ `eval[body[c], cons[parms[c], names[c]], cons[z, values[c]]]`

where `c = eval[fact, ((x) (fact) (q)), ((1) (C) (2))] = C`

and `z = evlis[(- x 1), ((x) (fact) (q)), ((1) (C) (2))] = (0)`

`= eval[B, cons[(x), ((fact) (q))], cons[(0), ((C) (2))]]`

`= eval[B, ((x) (fact) (q)), ((0) (C) (2))]`

`= eval[(if (eq x 0) 1 (* x (fact (- x 1)))), ((x) (fact) (q)), ((0) (C) (2))]`

`= eval[1] = 1`

So answer is

`2*1*1 = 2`