CSc 335 Sections M and R
Classes 3 and 4
February 1 and February 6 2024

; functions are values, too

(lambda (x) (* x x))

special form / reserved word

Scheme has first-class functions

lambda causes Scheme to create a closure

(

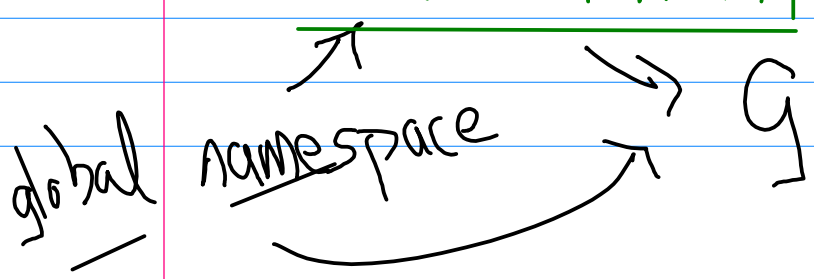
[You might enjoy searching the language Clojure]

That is, a function, along with certain environmental information

For example, in response to

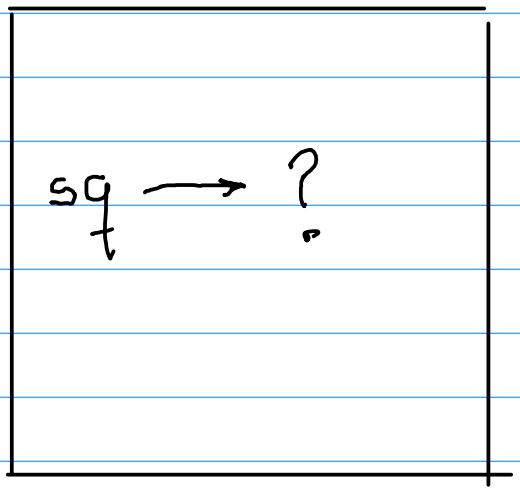
```
(define sq
   (lambda (x) (* x x)))
```

The system augments its starting environment as follows →



starting environment for scheme — everything predefined in scheme is installed here

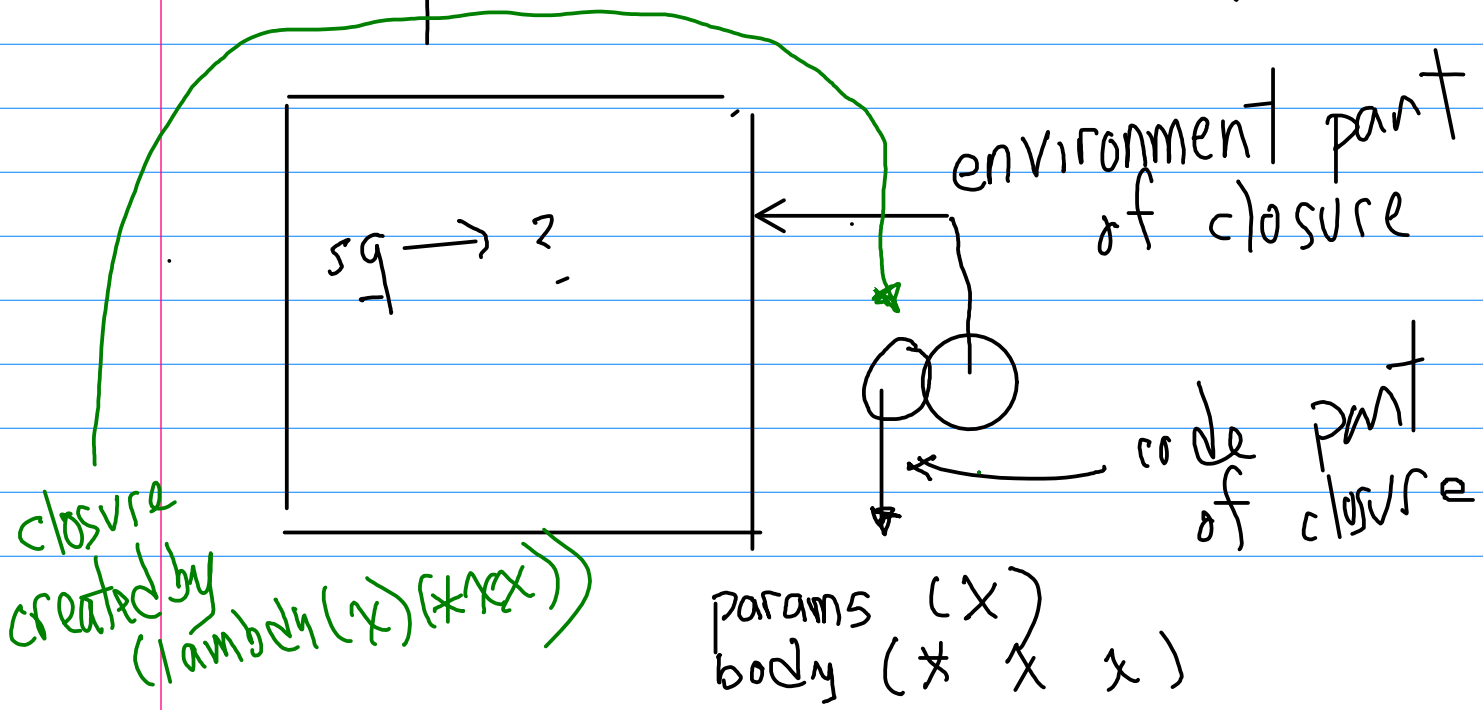
G



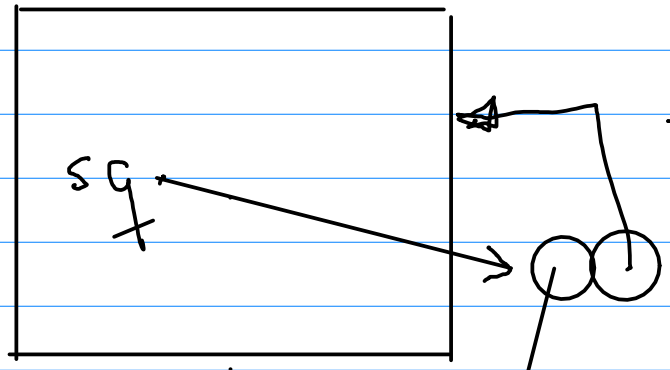

step 1 : install sq in G

(but we don't have a value for sq yet, hence the question mark)

step 2 : evaluate the lambda



Step 3: bind sq to the closure




The closure is a scheme object (much as the number 1 is a scheme object.

With this in place, what happens when we ask the system to evaluate

(sq 4) ?

Not sq(4)!

function name

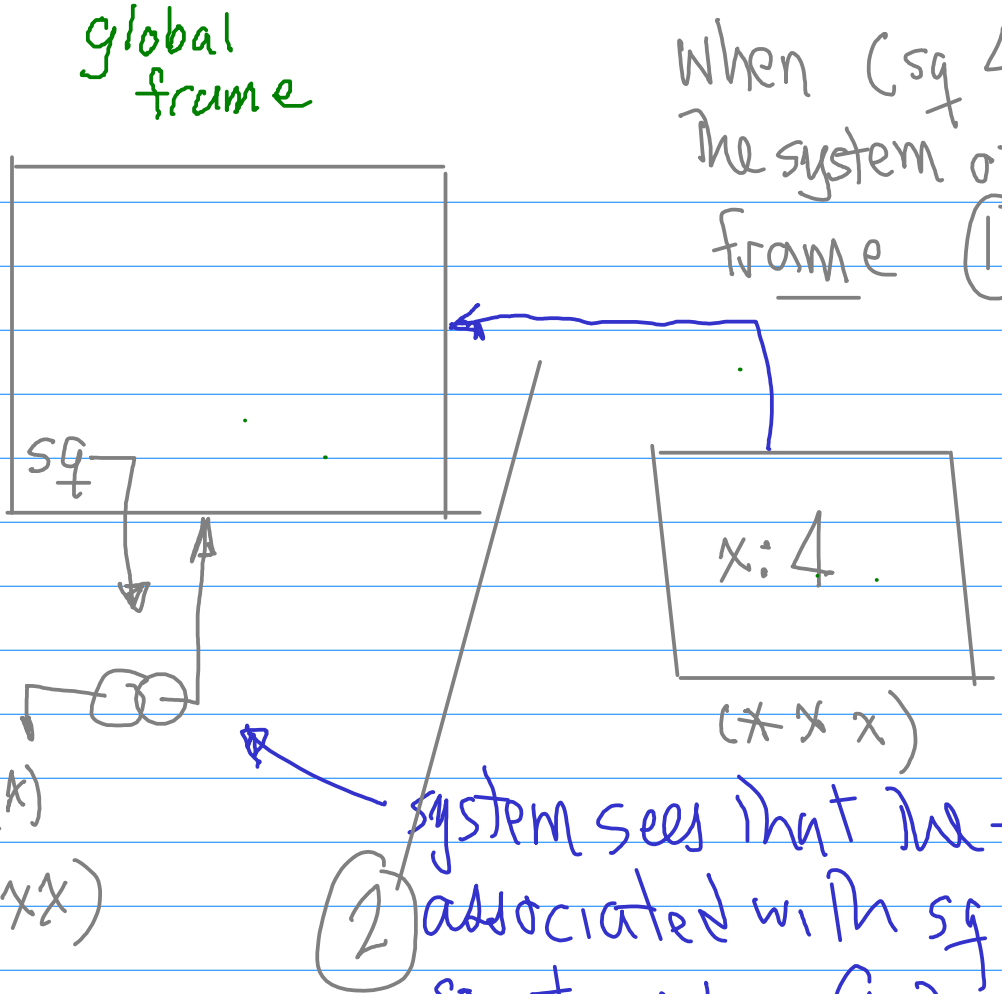
argument - ie- this is the ACTUAL parameter

for the def. given, scheme absolutely insists on exactly ONE argument

scheme has dynamic typing

G

an environment is a sequence of frames




When (sq 4) is requested the system opens a new frame (1)

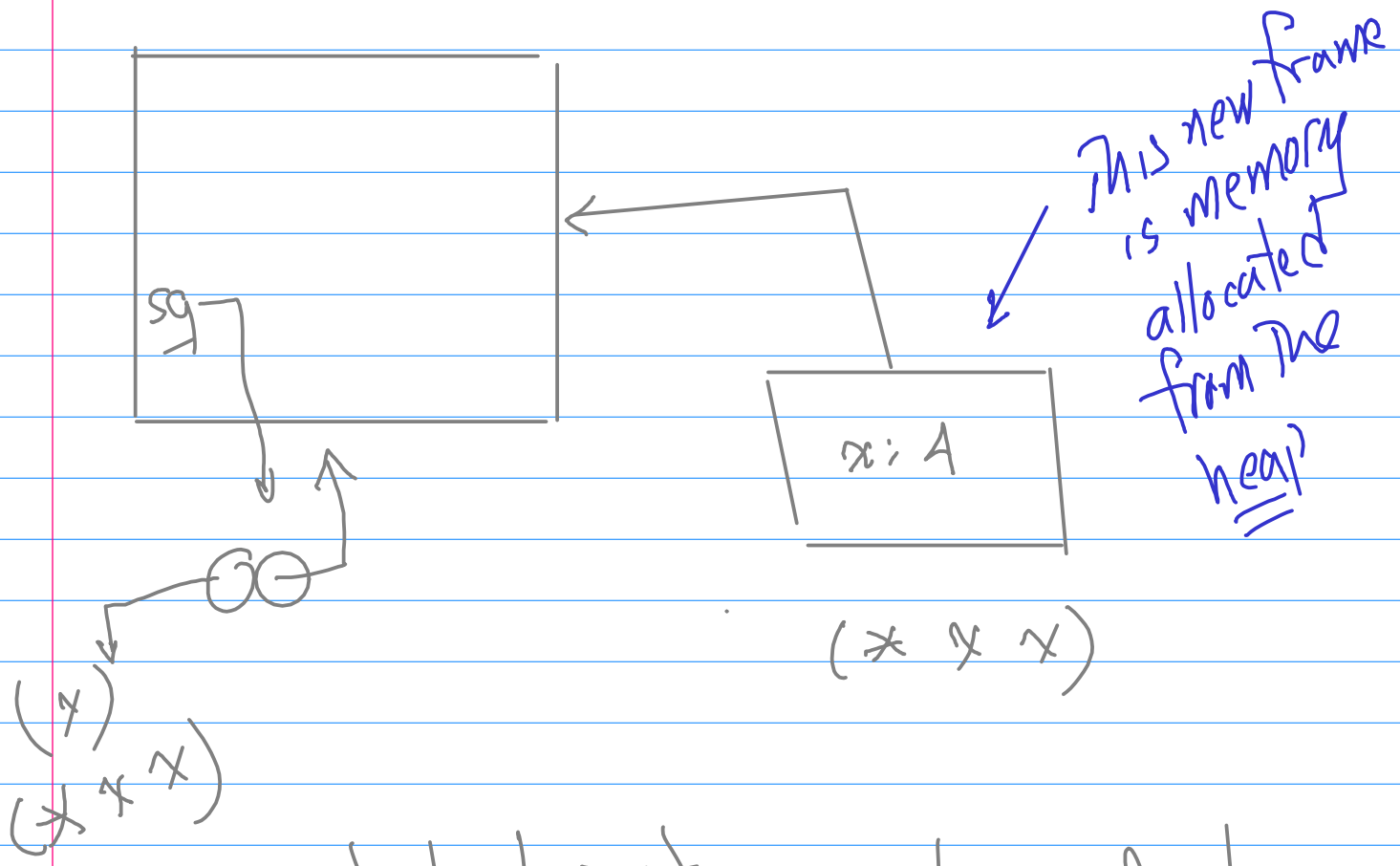
or parent frame

(2) system sees that the frame associated with sq is G, so it makes G the containing frame of this new frame

(3) in the new frame, the formal parameter x is bound to the actual parameter (in this case, 4) We say "x has value 4 in this frame"

(4) Next, scheme evaluates the body (* x x) in the new

frame




what does it mean to evaluate (* x x) in the new frame?

{ie, in the env. formed by the new frame and G, its parent.

We know from the scheme evaluation rule that the system needs to eval. * and x. To evaluate * you can think of the

system starting to look in the closest frame. But * is not defined in that frame (only x is). So the system goes to the parent frame and looks there for the definition of *.

Since in this case the parent frame is G, the scheme def. of * is found. So * is evaluated to the system mult. op.

What about x? Again, the system starts in the closest

frame in which x has been bound to 4.

So now (* 4 4) can be computed, and 16 returned.

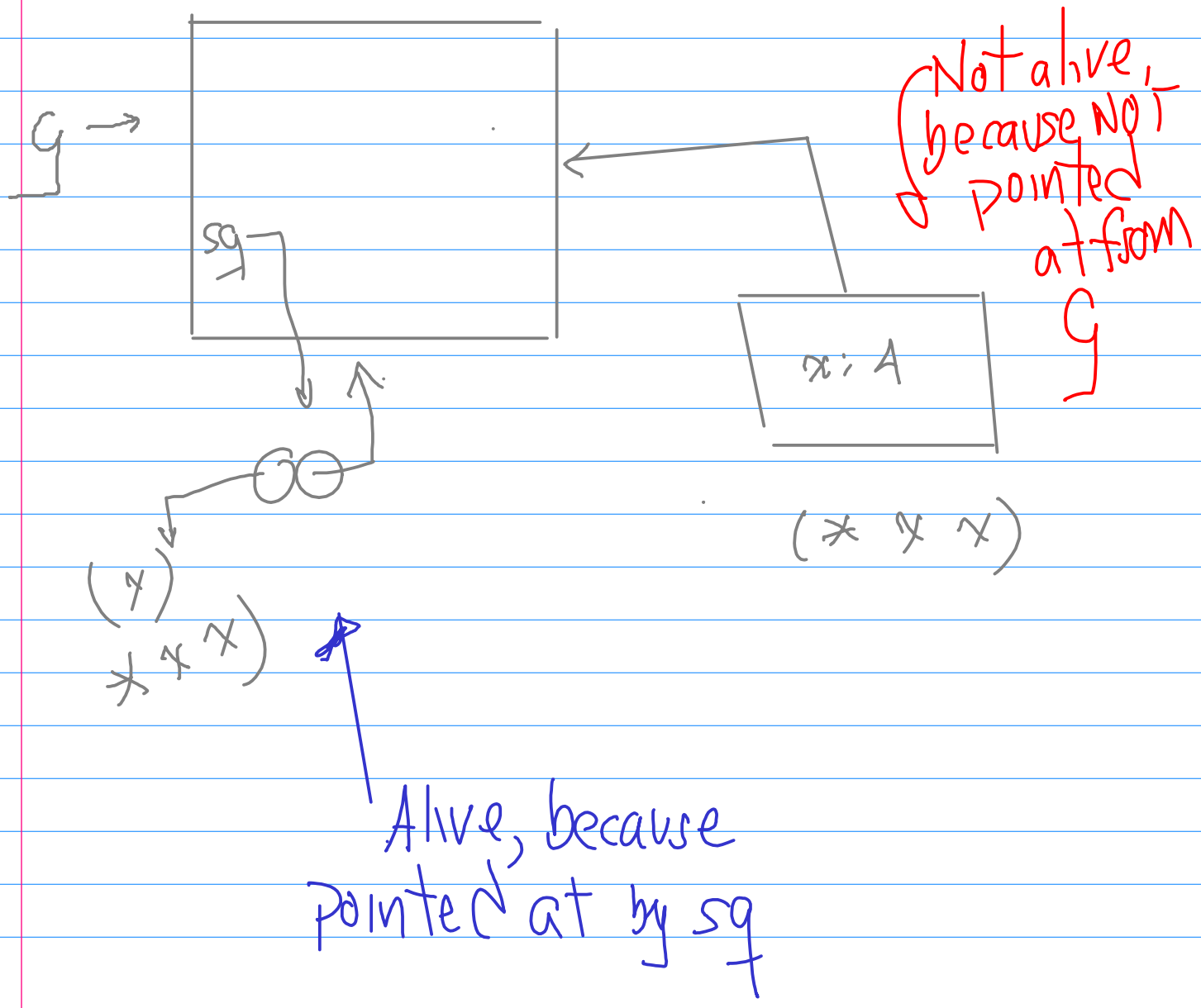
(5) Still not done ... we need yet to free up the memory allocated to the new frame. This is done by the garbage collector

(invented in the 1960s for LISP)

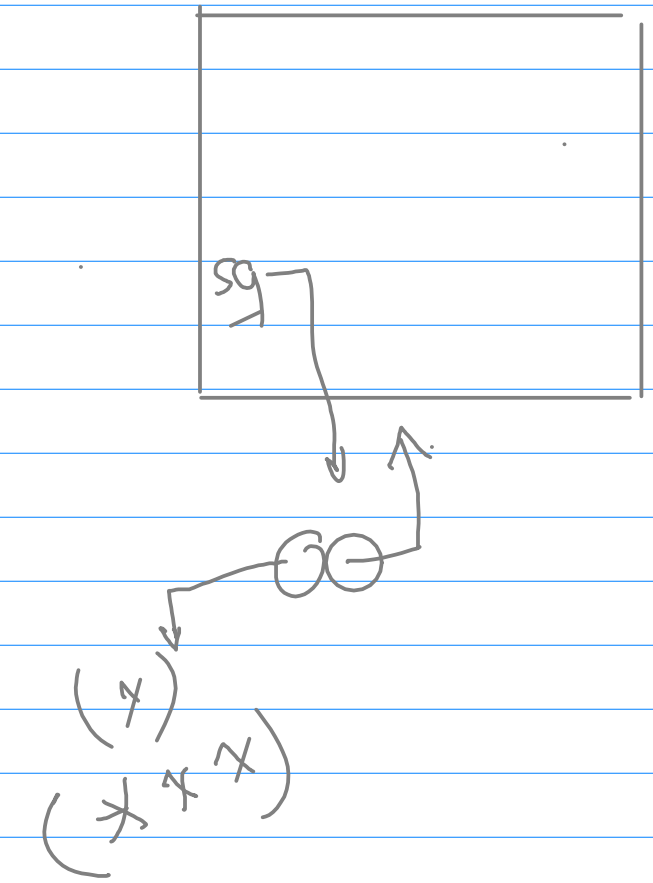
The idea is that storage which is no longer needed (ie, no longer "alive")

is returned to the heap.

How do we tell whether storage is still alive? Storage is live if it is pointed at (even indirectly) from the initial environment.



When the entire evaluation is done, the situation is




No concerns about memory leakage, as GC takes care of the problem.

---

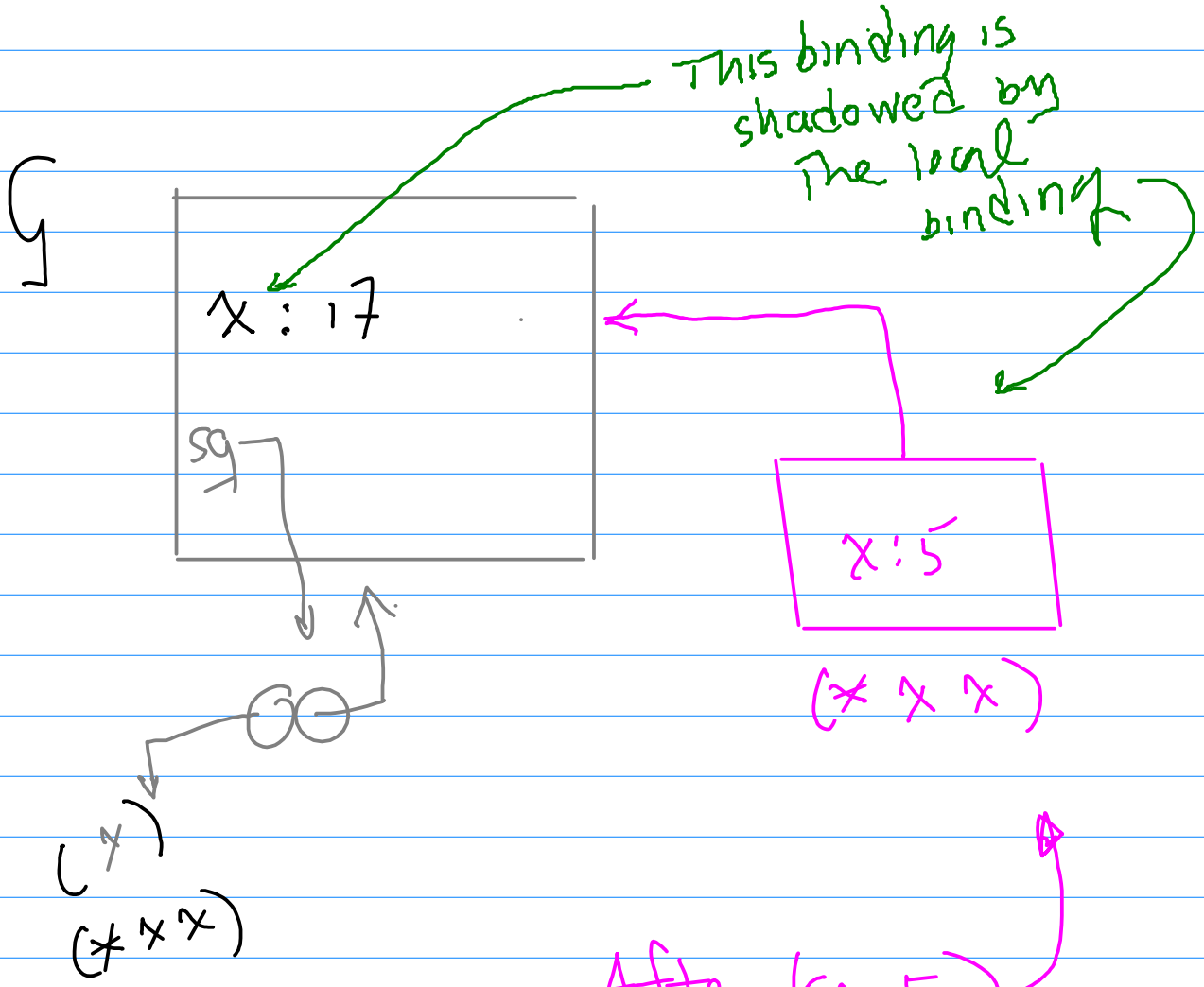
This evaluation model is described in Ch. 3 of A&S — EXCEPT they have introduced assignment by then. They call it the environment model of evaluation.

We can use this model to explain why (sq 5) does not change x if x has previously been defined.

Suppose (define x 17)
(sq 5)

and subseq. ask for x — you will see that it is still 17. Why?

Here's the situation after x and sq have been defined

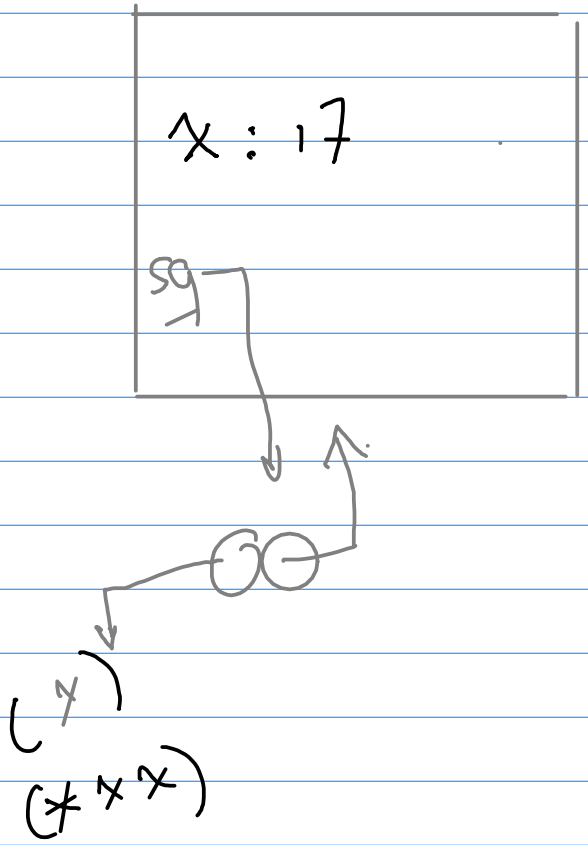



After (sq 5)

Again, * is found in G and x is found in the new frame

The system takes the first value of x that it finds as it works up the linked list of frames.

After computing 25, the new frame is garbaged — and we're left with




and of course when now we ask for the value of x, we get 17.

(the 5 is gone for good)

Abelson-Sussman format, which is — "under the hood" — the same as the lambda form

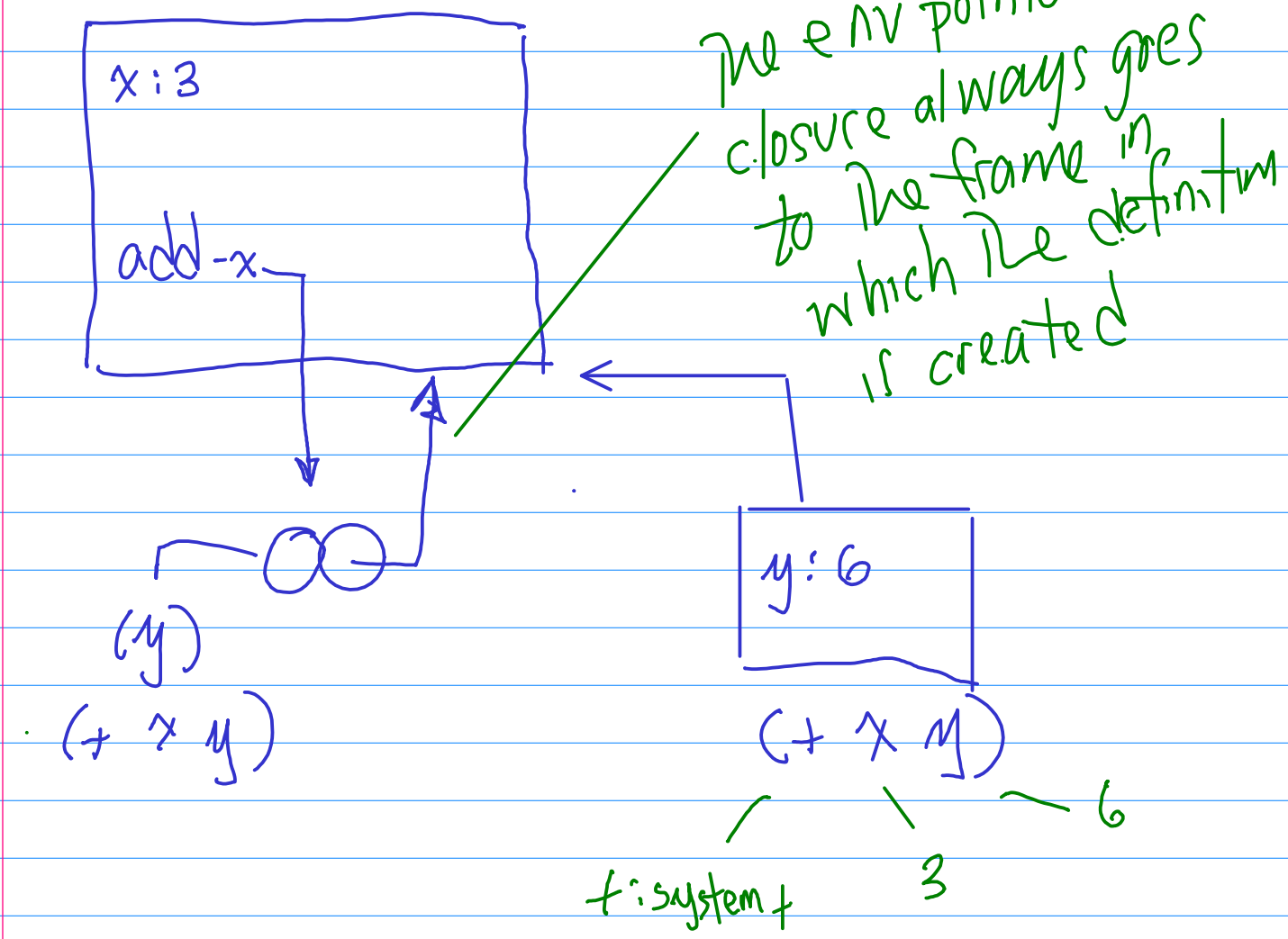
```
; some names are local, while others are global

(define x 3)

(define (add-x y)
  (+ x y))
```

x is free in the body of add-x

What happens when we have these definitions, and then ask for (add-x 6) ?




the env pointer of the closure always goes to the frame in which the definition is created

Student Question: What if add-x were defined

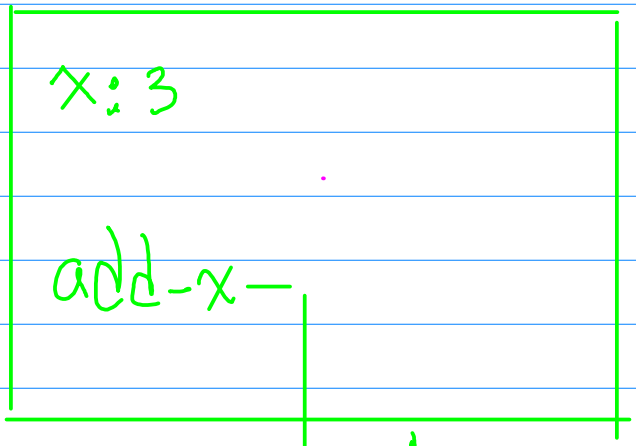
```
(define (add-x x y)
  (+ x y))
```

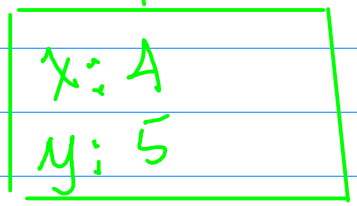
?

Same as

```
(define add-x
  (lambda (x y)
    (+ x y)))
```

Consider (add-x 4 5)




(x y)
(+ x y)

(+ x y)
4 5

Note that you may NOT call this new version with just a single

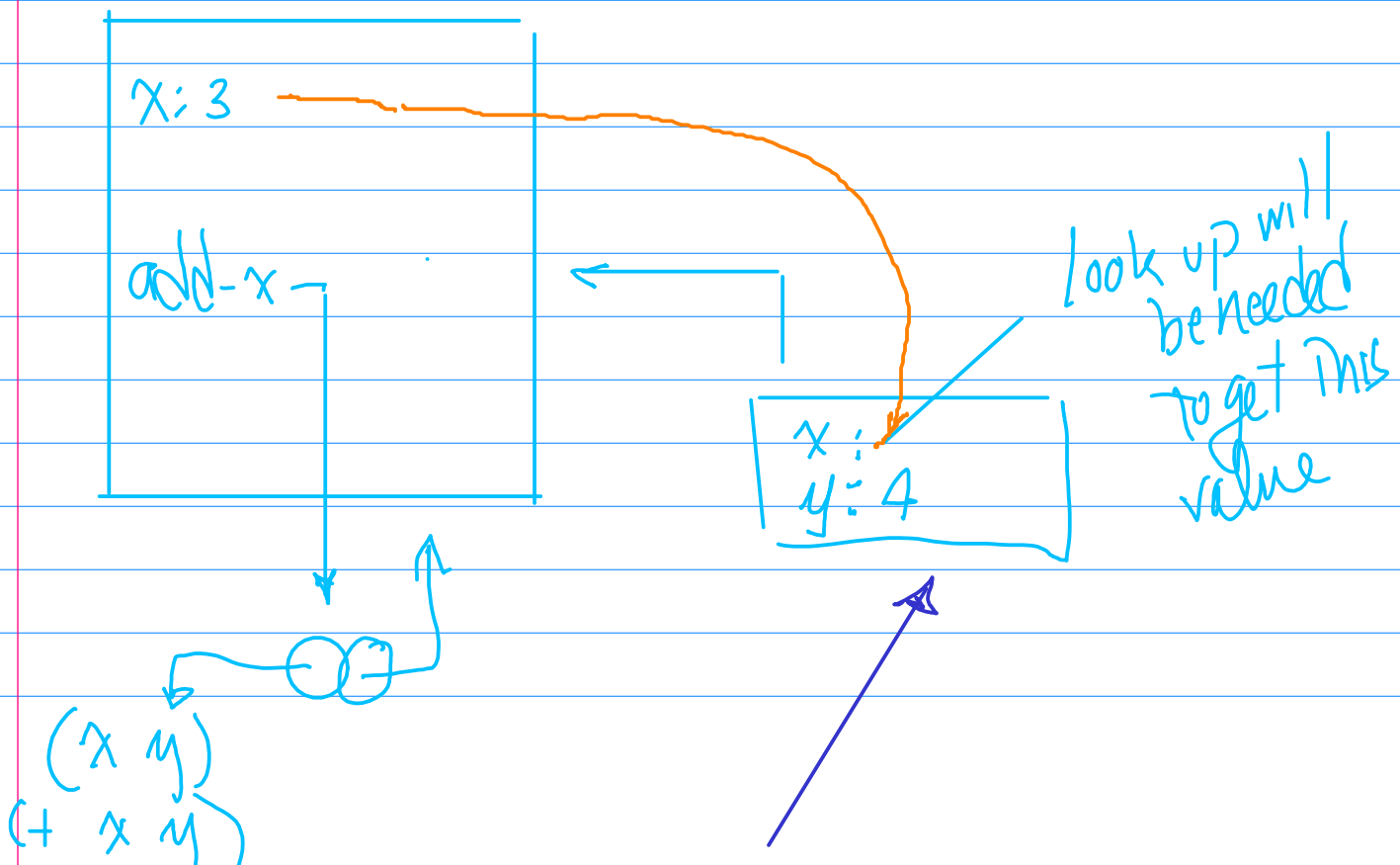
Parameter, because Scheme will regard that as a type error

What about

```
(define x 3)
(add-x x 4)
```

?

In this case the system would eval x to 3



At this point, schemes default parameter passing mechanism is call-by-value

(a local copy of the param. value is created)

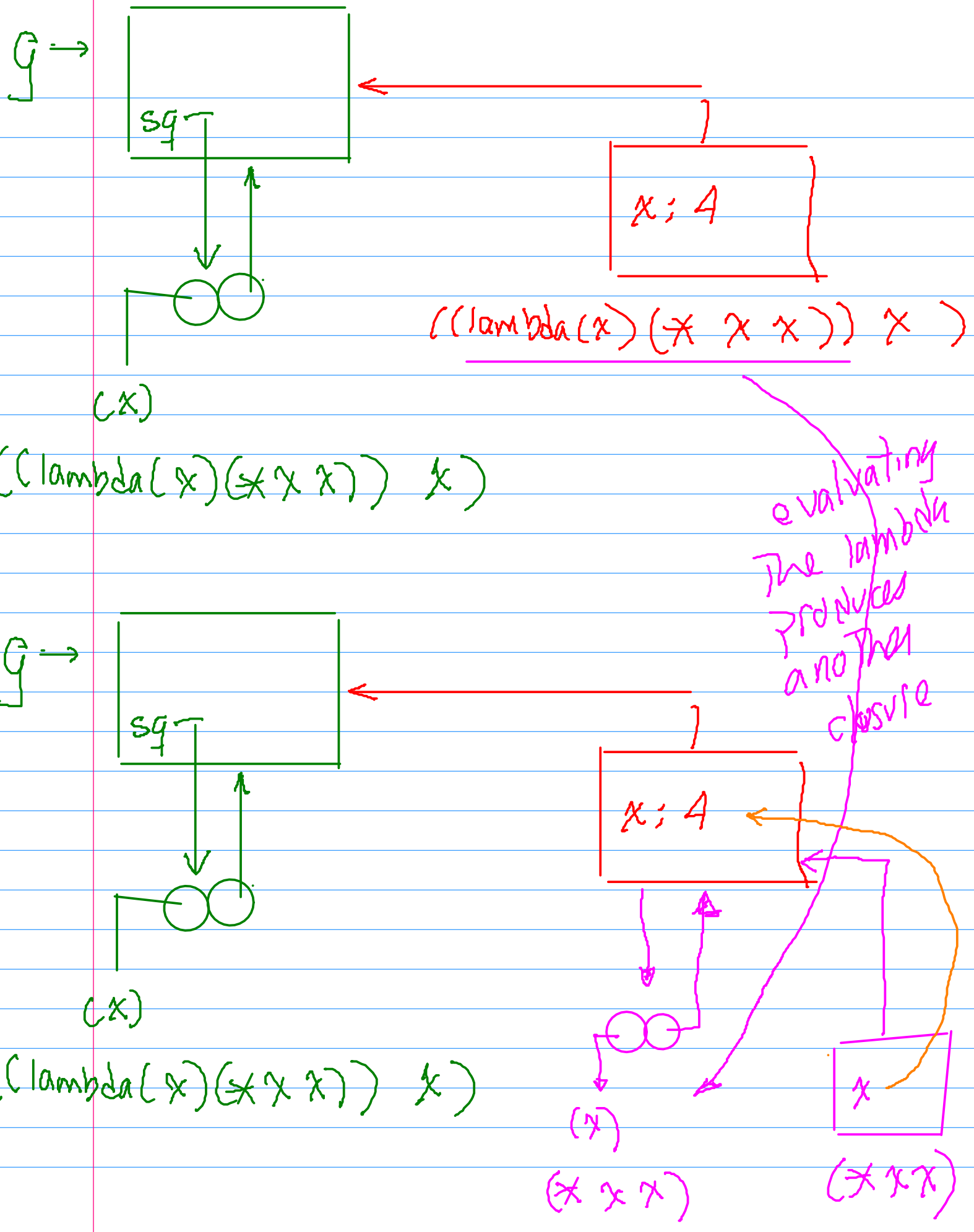
Student Question

What about

```
(define (sq x)
  ((lambda (x) (* x x)) x))
```

Consider (sq 4)

```
(define sq
  (lambda (x)
    ((lambda (x) (* x x)) x)
    ))
```


g
sq
x : 4
(x)
((lambda(x)(* x x)) x)
((lambda(x)(* x x)) x)
evaluating the lambda produces another closure
g
sq
x : 4
(x)
(* x x)
x
(* x x)
((lambda(x)(* x x)) x)

# Function Composition

One imagines wanting a function cube – one idea might be

```
(define (cube x)
  (* x x x))
```

what if $x^{47}$ were wanted?

but another (better) would be to break the function down as

```
(define (cube x)
  (* x (square x)))
```

even without having first implemented the square function.

(notice the divide & conquer thinking)

Another example:

```
; functions may occur in other functions - for example

(define (sum-of-squares x y)
  (+ (square x) (square y)))


; we can use sum-of-squares as a building block in constructing
; further procedures

(define (f a)
  (sum-of-squares (+ a 1) (* a 2)))


(f 5)
```

These occurrences of a are said to be free in the body of f.

but they are bound in f to the formal parameter or

We can calculate (f 5) using either what is called the substitution model or the model introduced last time (the environment model)

see ch3 in A&S

The substitution model: to evaluate a function call - s (f 5) - substitute the value of the actual parameter for every free occurrence of the corresponding formal parameter in the function body

renew quantifiers ($\forall, \exists$) in your discrete math Predicate Logic

For (f 5):

① substitute 5 for the free occurrences of a in (sum-of-squares (+ a 1) (* a 2))

Eager evaluation (applicative order)

gives

(sum-of-squares 6 10)

↑ we sub for the parameters to sum-of-squares before we sub for sum-of-squares

② now do the same thing for sum-of-squares

(+ (square 6) (square 10))

③ now do the same thing for square

(+ 36 100)

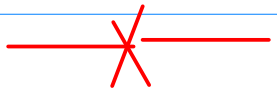
④ compute: 136

The substitution model is valid ONLY if assignment is absent — consider

$$x = x + 1$$

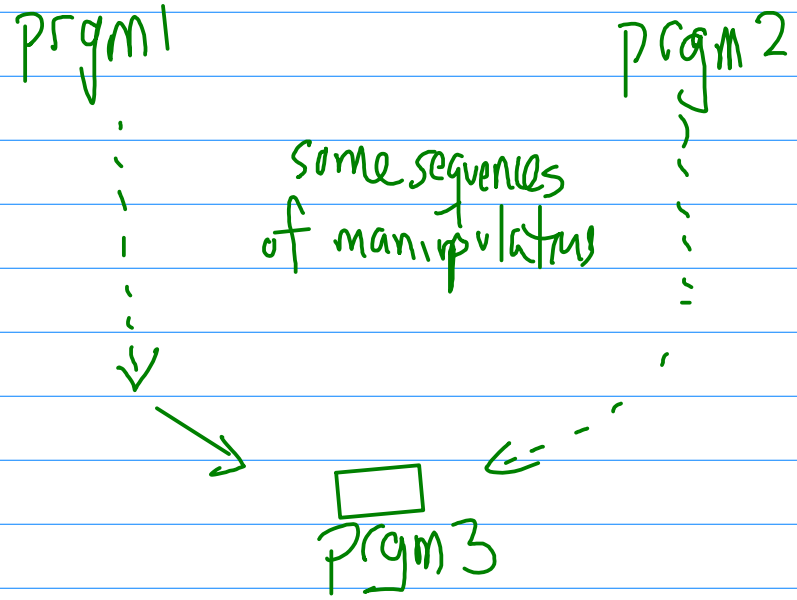
we substitute 6 for x and get

$$6 = 7$$

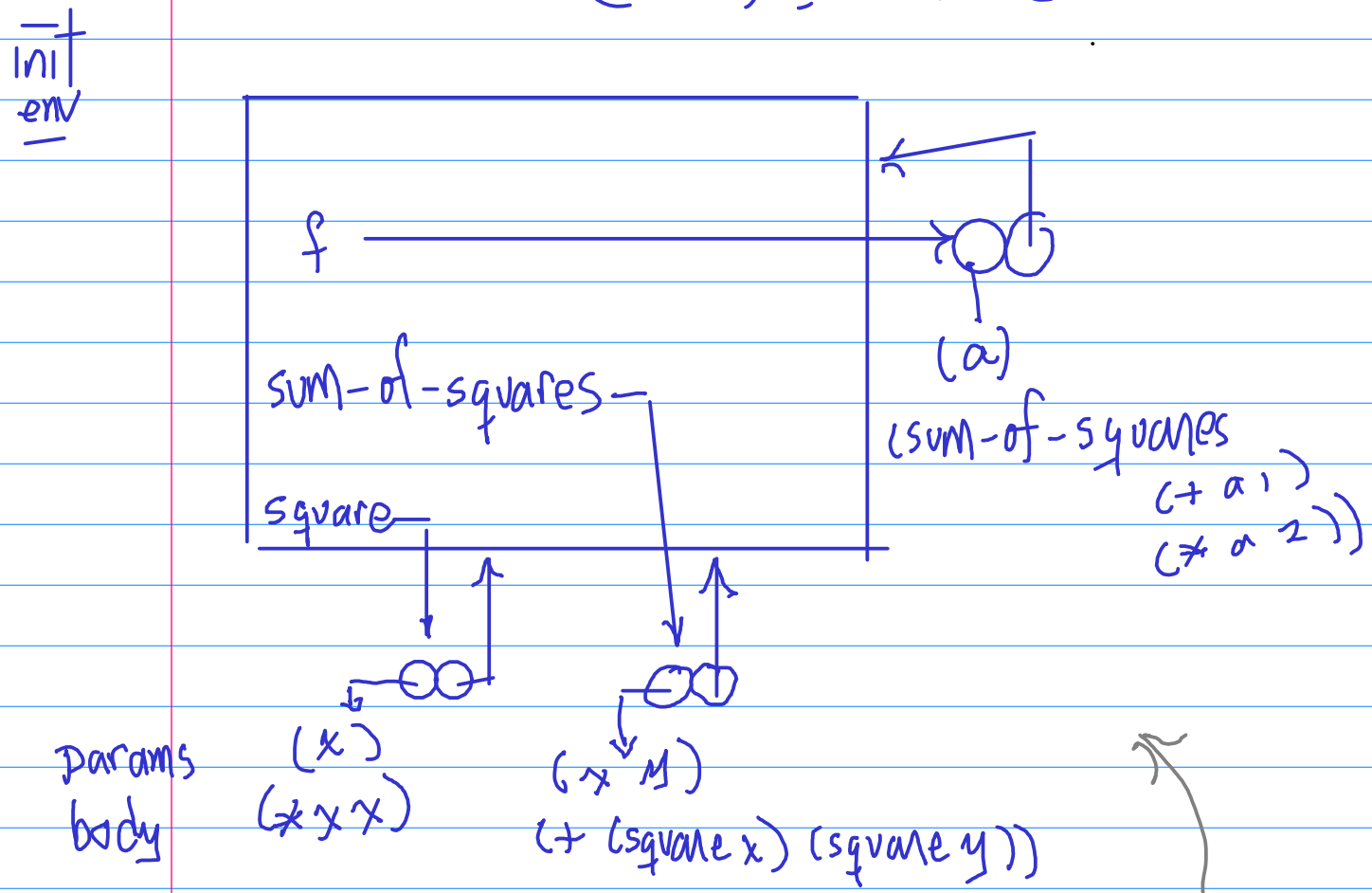


visual symbol for logical contradiction.

The substitution model should remind you of (pre-calc) algebra — and indeed, it suggests the possibility of an algebra of programs which would allow me to work with programs as easily as one works with algebraic equations. Eg: do prgm1 and prgm2 compute the same values?



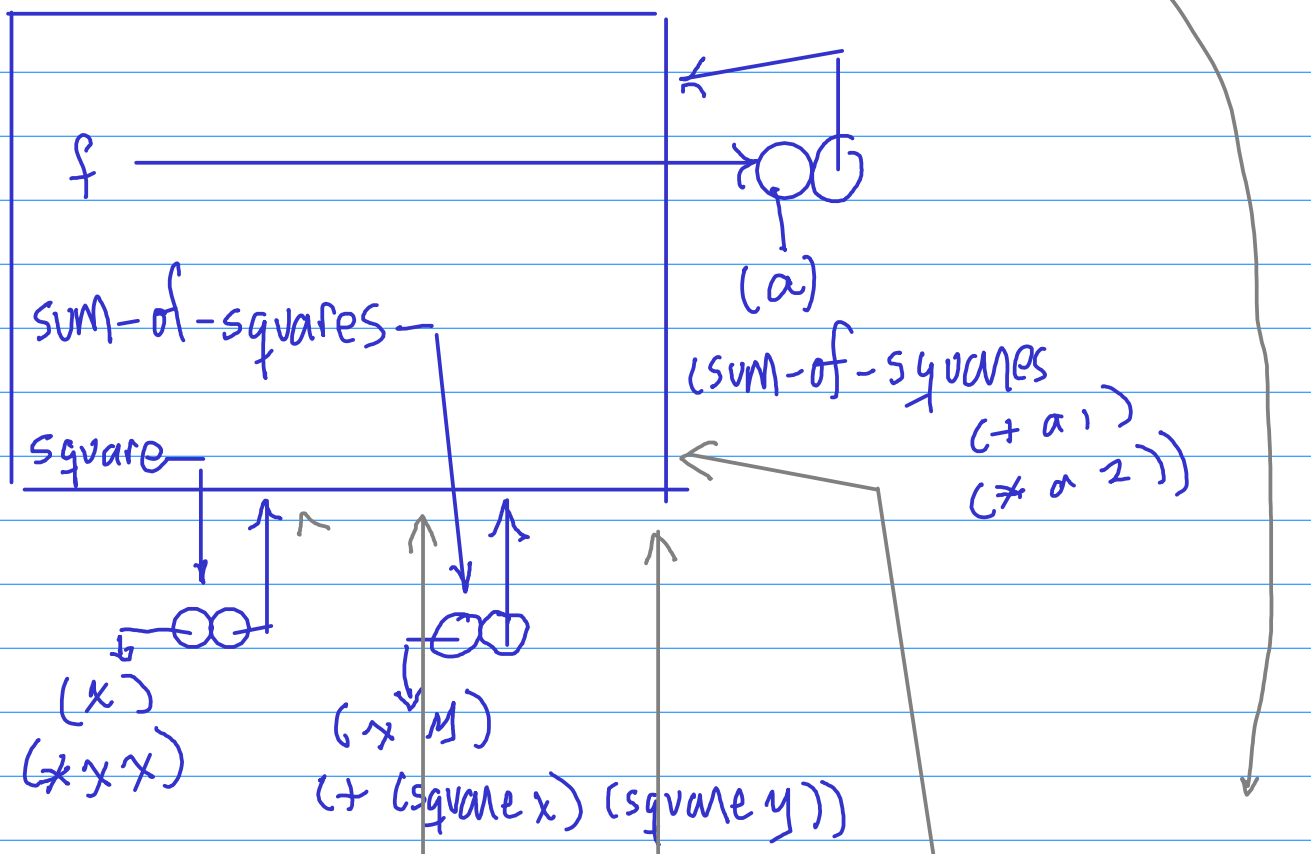
Can we at least start the environment model evaluation of (f 5)? Sure...




After the functions have been defined in a (fresh) initial environment

We call (f 5)

first step: using the closure for f




create new frame

a: 5

(sum-of-squares 6 10)

second step:
eval call to sum-of-squares

x: 6
y: 10

(+ (square 6) (square 10))

third step
eval calls to
square

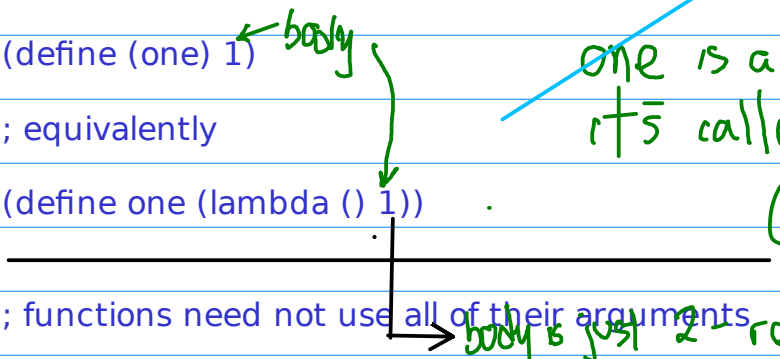
x: 6
(* x)

x: 10
(* x x)

# Two useful observations on functions

```
; functions can also have 0 arguments
(define (one) 1)
; equivalently
(define one (lambda () 1))

; functions need not use all of their arguments
(define (two x y z) 2)
; equivalently
(define two (lambda (x y z) 2))
```




illustrates the empty (or trivial) substitution

one is a constant — every time it's called, it returns 1.

(Work this out with both eval models)

# Applicative and Normal Order

```
; let's recall the earlier example

(define (f a)
  (sum-of-squares (+ a 1) (* a 2)))
```

```
; using applicative order, the evaluation
; of the call (f 5) proceeds as

; (f 5)

; ((sum-of-squares (+ a 1) (* a 2)) 5)

; (sum-of-squares (+ 5 1) (* 5 2))

; (sum-of-squares 6 10)

; (+ (square 6) (square 10))

; (+ (* 6 6) (* 10 10))

; (+ 36 100)

; 136
```

```
; using normal order, the computation
; proceeds differently:

; (f 5)

; ((sum-of-squares (+ a 1) (* a 2)) 5)

; (sum-of-squares (+ 5 1) (* 5 2))

; (+ (square (+ 5 1)) (square (* 5 2)))

; (+ (* (+ 5 1) (+ 5 1)) (* (* 5 2) (* 5 2)))

; (+ (* 6 6) (* 10 10))

; (+   36     100)

; 136
```

In this case, applicative and normal order give the same result, but this is not always the case.

; applicative and normal order evaluation do not always give the same result

; consider

(define (p) (p))

(define (test x y)
  (if (= x 0)
      0
      y))

; what will happen if (test 0 (p)) is evaluated using applicative order? why?
; what will happen if it is evaluated using normal order?

; you can stop an infinite loop under DrRacket by using the red Stop button
; at the top right of your screen

; remark

(if #t 0 (p))

; returns 0 -- because if is a special form, the infinite loop (p) is never evaluated

; to drive this point home, suppose that instead of if, we defined our own function, ne

(define (new-if predicate then-clause else-clause)
  (cond (predicate then-clause)
        (else else-clause)))

(new-if (> 2 0) 3 4) ; => 3

(new-if (> 2 3) 3 4) ; => 4

; this seems to work! But should we try ...

(let ((x 0))
  (new-if (= x 0)
      0
      (p)))

; we find that we have an infinite loop, precisely because the standard evaluation
; rules apply to the user-defined function new-if: new-if is not a special form

equivalently (define (p) (lambda () (p)))
observe that test, as a user-defined function, will follow applicative order
with this call, control never reaches the if...
as scheme gets stuck trying to eval (p)

normal order, (p) is never evaluated so the call would return 0

can't test without a normal order evaluator

if and cond are "partially" lazy in that although the predicate is always evaluated only one of the then- and else- clauses is evaluated, depending on the value of the predicate.

attempt at user-defined if

predicate is a boolean-valued function

notice that we are passing a function as a parameter.

applicative order forces both to be evaluated.

# Boolean Functions and Conditionals

```
(define (myabs x)
  (cond ((> x 0) x)
        ((= x 0) 0)
        (else (- x))))

(myabs -4)

; another way

(define (myabs x)
  (cond ((< x 0) (- x))
        (else x)))

; another way

(define (myabs x)
  (if (< x 0)
      (- x)
      x))


; >, =, < are primitive predicates

; Scheme also supplies logical connectives such as and, or, not

(define x 6)

(and (> x 5) (< x 10))

(not (> x 4))

#f

(not #f)

(= x 6)
```

cond has its own evaluation rule →

minus is here unary negation op

>= for ≥

<= for ≤

necessary for formulating more general predicates

and is NOT a scheme function - it is instead a macro

conjuncts are evaluated left to right, with immediate exit when one of them tests to #f false

This evaluation order is NOT part of the language standard

general form:
cond followed by a list of question-answer pairs (p e)
p: "predicate"
e: "expression"

optional else

; note that cond and if do not obey the standard evaluation rule -

; (cond (p1 e1) (p2 e2) ... (pn en) (else E)) is evaluated as follows:
;  evaluate p1 -- if #t, then evaluate e1 and return this value (and exit the cond)
;  otherwise evaluate p2 -- if #t, evaluate e2 and return this value
;  otherwise evaluate p3 -- if #t, evaluate e3 and return this value

;  and so on.  If none of the pi is #t, evaluate E and return this value

ONLY eval the answer of the associated predicate is true.

; similarly, (if p1 e1 e2) evaluates e1, but not e2, if p1 is #t; it
; evaluates e2, but not e1, if p1 is #f

"partially lazy"

; it is interesting to note that cond and if forms can be evaluated
; by themselves - they do not need to be embedded in a function

(define x 6)

(cond ((> x 6) 1)
    ((< x 6) 2)
    (else 0))

cond and if can occur alone as input to scheme

(if (= x 6) 2 3)

"if" is partially lazy in the same way

; and even such expressions as

(+ (cond ((> x 6) 0)
      (else 1))
  2)

and can also occur in ANY expression,

eg)

A more interesting example:

make sure you've fully understood the scheme evaluation rule!

(define (a-plus-abs-b a b)
  ((if (> b 0) + -) a b))

We can return functions as values in Scheme.

```
(a-plus-abs-b  2  3) = (+ 2 3) = 5
(a-plus-abs-b  2 -3) = (- 2 -3) = 5
```

---

* It is instructive to note that and and or can be defined in terms of cond.

and in terms of cond?

(cond (x y)) $\stackrel{?}{=}$ (and x y)

discrete math!!

let's look at the truth tables

| x | y | (cond (x y)) | (and x y) |
|---|---|---|---|
| T | T | T | T |
| T | F | F | F |
| F | T | unspecified | F |
| F | F | unspecified | F |

So the functions are NOT the same - though we could say that the restrictions of the functions defined by (cond (x y)) and (and x y) to all pairs whose first elt is T are the same.

Can we remove this restriction? Easy enough:

```
(cond (x y)
      (else #f))
```

You can check that this is precisely (and x y)

Suggested HW: ① Show how OR can be implemented using cond.

② Think about whether NOT could be implemented using cond.