(ref. to Abelson & Sussman)

```
; Section 1.3.4  Procedures as Returned Values

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (make-addConstant x)
  (lambda (y) (+ x y)))
```

(define make-addConstant
 (lambda (x)
  (lambda (y) (+ x y))))

```
((make-addConstant 4) 5)
```

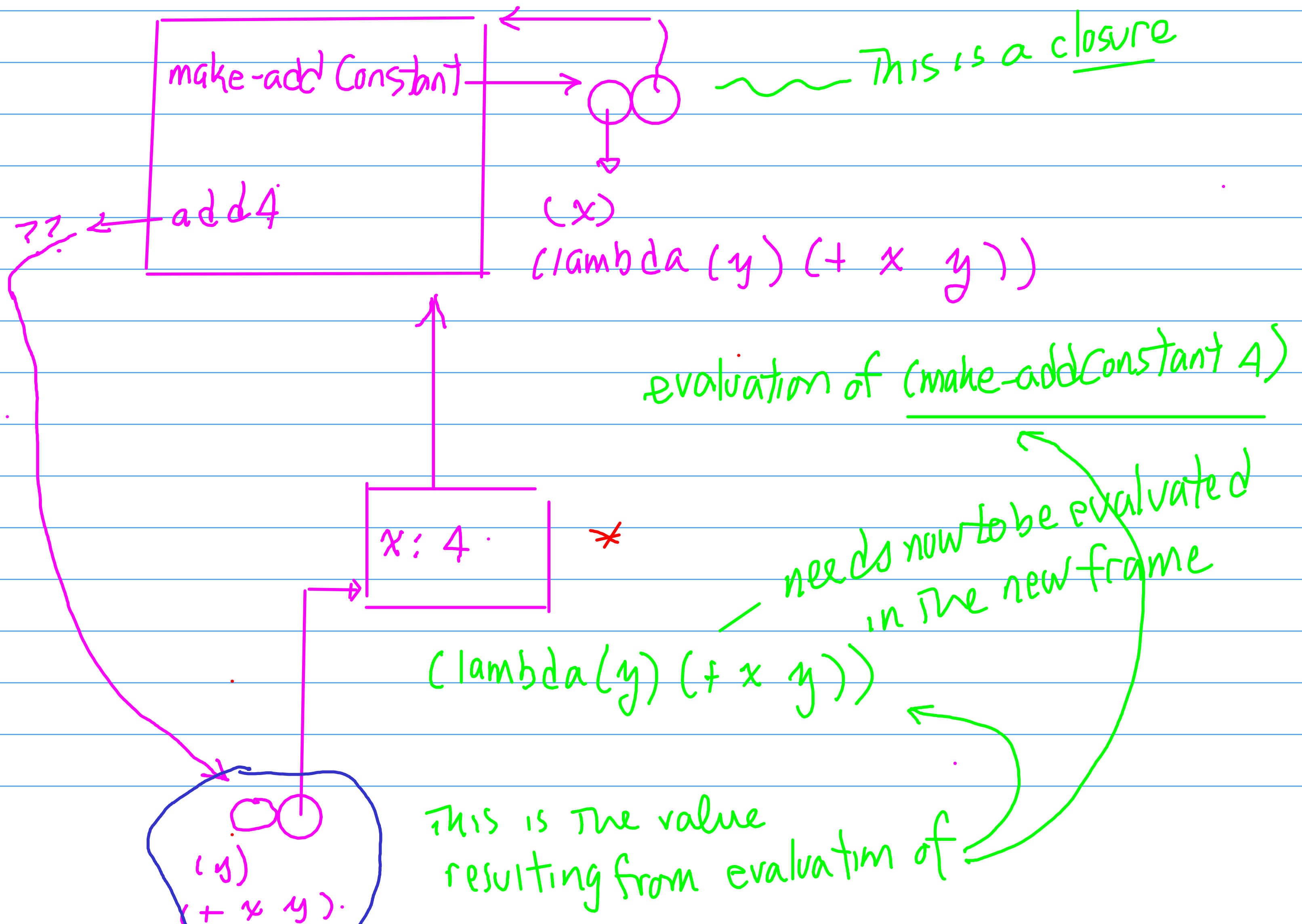
9

(lambda (y) (+ 4 y))

```
(define add4 (make-addConstant 4))

(add4 3)
```

environment model simulation

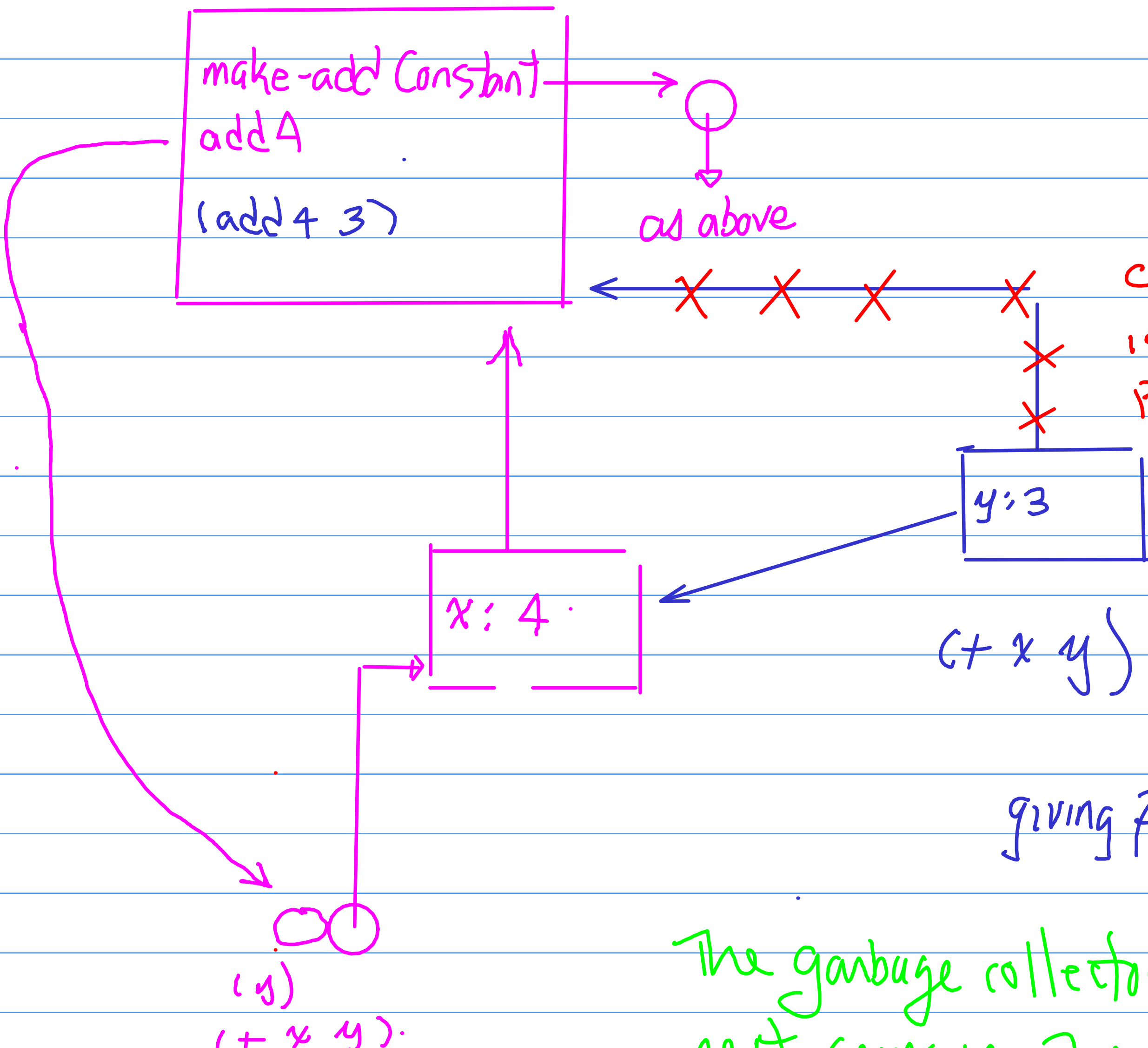


Might expect the frame * would be garbaged once the evaluation is complete — but this does not happen because * is (indirectly) pointed at by an object — add4 — at the top level. We say that * is <u>live</u>

Recall that an environment is defined to be a <u>sequence of frames</u> — so the add4 object (or just add4) has a 2-frame environment.

Let's figure out what happens when we evaluate
(add4 3).

A look at (The simulation of) (add4 3) :



containing frame is The frame pointed at by The closure object, not The frame in which The call is made.

giving 7.

The garbage collector will next remove The frame in which y is bound to 3.

```
(define (sigma a b term)
  (cond ((> a b) 0)
        (else (+ (term a) (sigma (+ a 1) b term)))))
```

{Haskell Curry was a logician}

```
; curried form of sigma
```

(currying a function)

```
(define (curried-sigma term)
  (define (sum-term a b)
    (cond ((> a b) 0)
          (else (+ (term a) (sum-term (+ a 1) b)))))
  sum-term)
```

def part

body

internal function definition

body of the function curried-sigma. The scope of functions defined in the def. part is precisely the body + the def part

```
((curried-sigma (lambda (x) x)) 1 10)

(define sum-of-squares
  (curried-sigma (lambda (x) (* x x))))

(sum-of-squares 1 10)
```

term

So the value returned is the sum-term function with (lambda (x) (* x x)) substituted for term.

```
; contrast to

(sigma (lambda (x) (* x x)) 1 10)
```

The uncurried sigma, in which the term parameter must be passed explicitly.

Interesting for a couple of reasons — ① it allows another level of code reuse; ② it can facilitate efficiency by setting up 'partial compilation'; ③ it suggests that — in principle — we only need single-argument functions

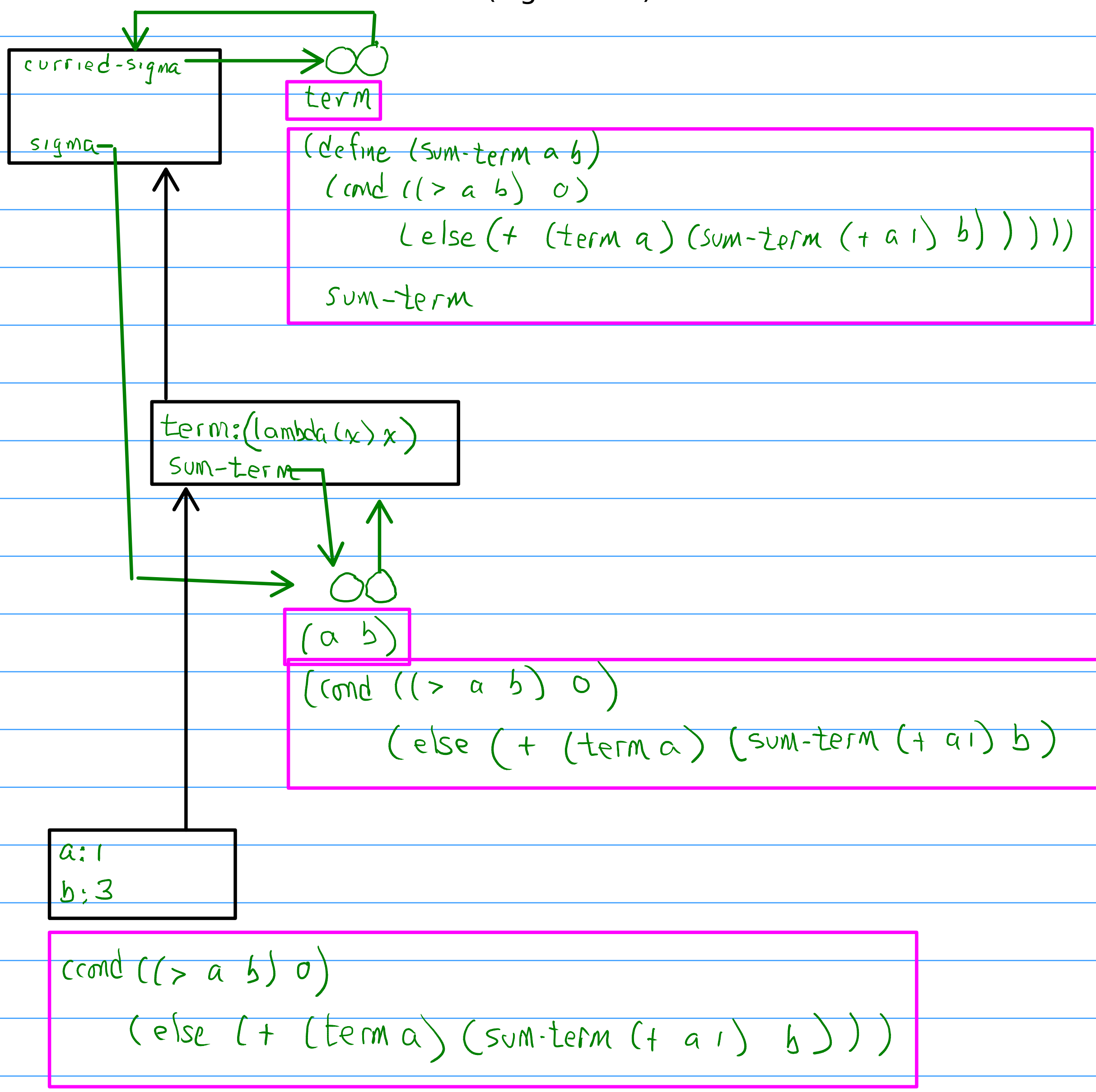
functions of n params to functions of n-1 params

"function specializing"

```scheme
(define (curried-sigma term)
  (define (sum-term a b)
    (cond ((> a b) 0)
          (else (+ (term a) (sum-term (+ a 1) b)))))
  sum-term)

(define sigma (curried-sigma (lambda (x) x)))

(sigma 1 3)
```



suggested exercises

→ cook with curry on your own → ie →
Try your hand at currying a function or two

→ See if you can work out the environment diagram simulation for curried-sigma close with an example from A&S hinting at the usefulness of having functions as values.

```
(define (deriv f)
  (let ((dx .00000001))
    (lambda (x) (/ (- (f (+ x dx)) (f x))
               dx))))

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

((deriv cube) 5)
```

The value returned is

$$\left(\mathrm{lambda}(x)\ \frac{(f(x+dx) - f(x))}{dx}\right)$$

where dx is $.00\text{----}01$

So (deriv cube) gives the function of x

$$\frac{(\mathrm{cube}(x+dx) - \mathrm{cube}(x))}{dx}$$

which you recognize as the difference quotient for cube.

well, we can't take limits

Recall: deriv of cube at the point x

$$\lim_{\Delta x \to 0} \frac{\mathrm{cube}(x+\Delta x) - \mathrm{cube}(x)}{\Delta x}$$

in calculus, derivatives are functions. and scheme is a language in which this can be expressed directly.