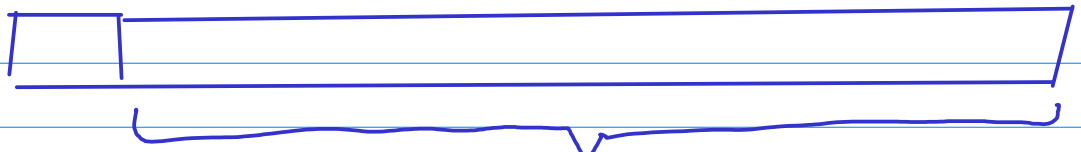
1. Pertaining to basic set operations

Lest you think everything in this direction is boring — even for finite sets — consider the problem of computing the intersection of a collection of sets (rep. as lists without repetition) in which any of the sets might be empty.

L

Basic Design (recursive)



recursively compute
(intyset (cdr L))

and then
(intersect (car L) ↑)

In vanilla functional scheme — how would we arrange for early exit should (say) the 128th of the 129 sets input) turns out to be '() ?

This is a motivating example for learning about

continuations

Scheme has 1st-class continuations — was first for this and still one of very few languages to offer this kind of power.

The scheme feature call/cc (short for call-with-current-continuation) allows for early exit

...

and all kinds of other very cool stuff.

The continuation is perhaps the most powerful flow structure in programming.

Good first reference →
The Seasoned Schemer

Good second reference —

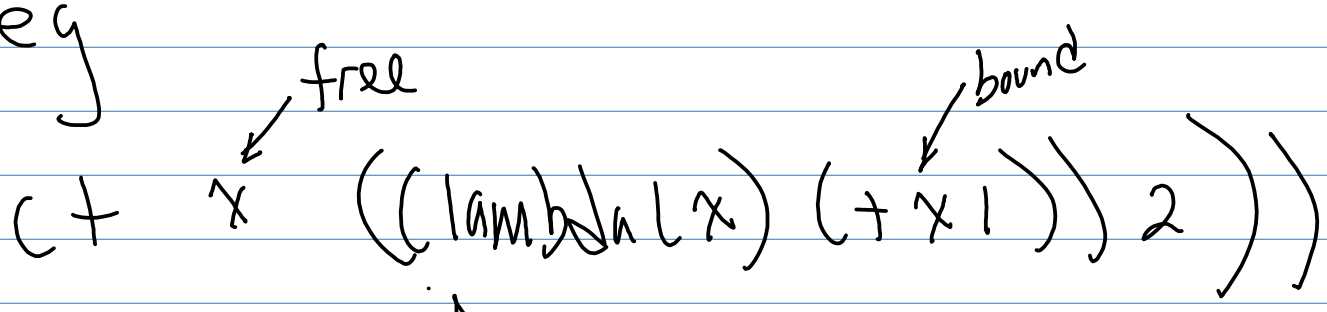
EOPL (second Ed)

↑ also talks about cps

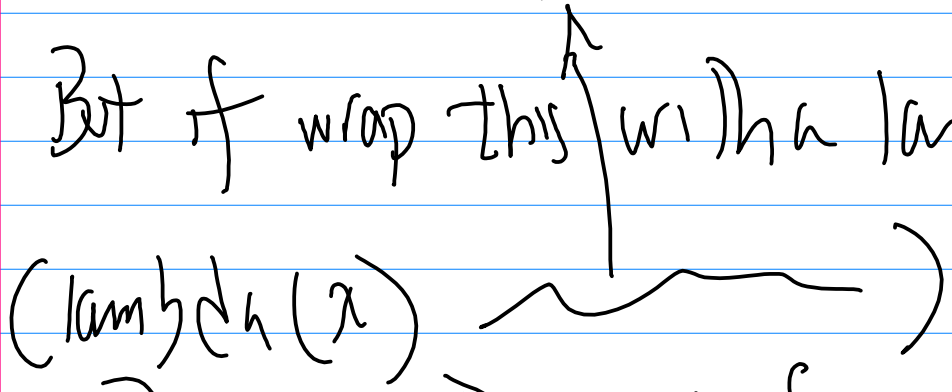
Continuation-passing-style

(2) its routine for a variable to be both free and bound in a single exp.

eg



But if wrap this with a lambda (x)



Then are then NO free variables.

```
(lambda (x)
  (+ x ((lambda (x) (+ x 1)) 2)))
```

↓ α-variation (change outer x to y)

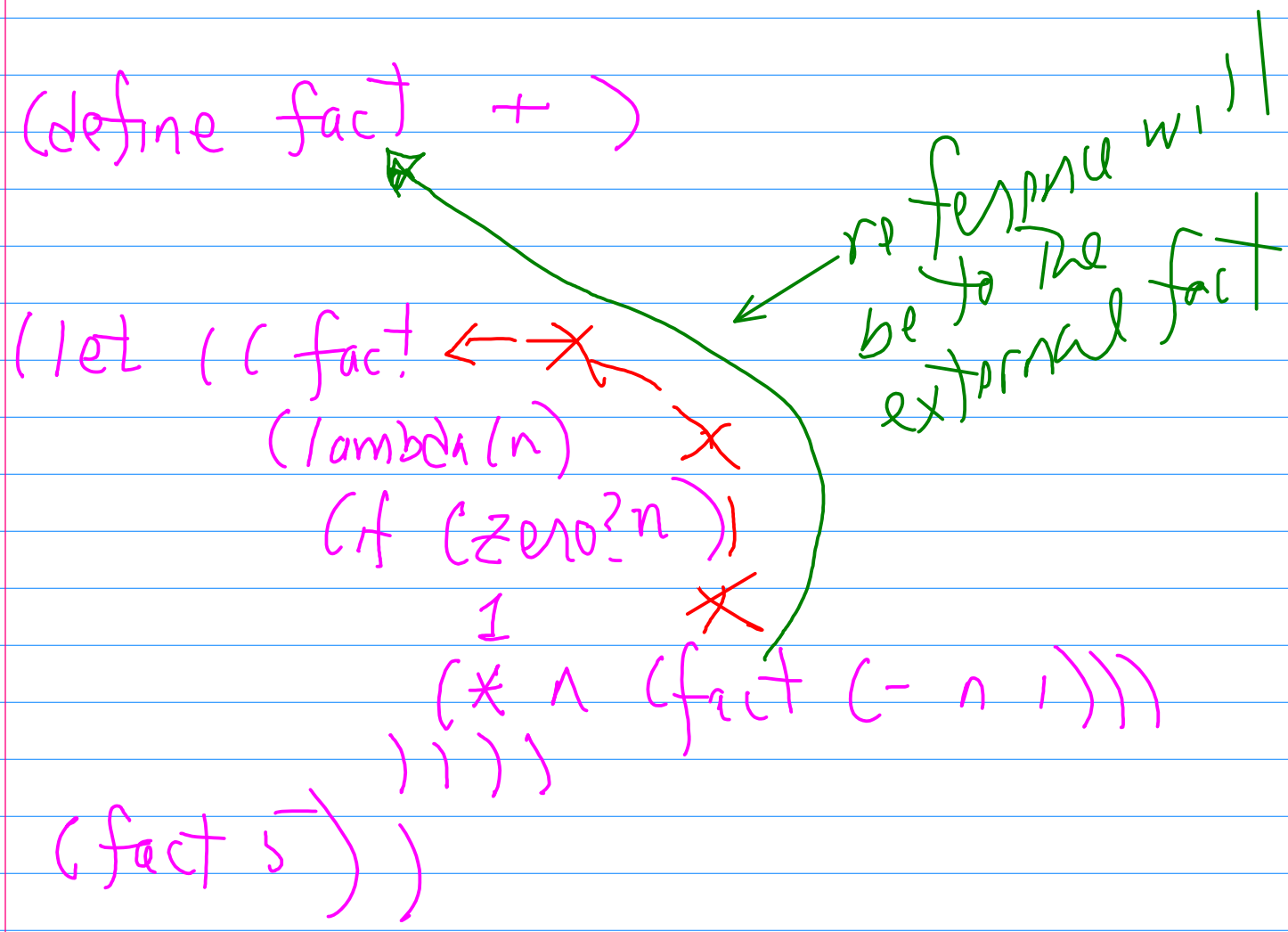
```
(lambda (y)
  (+ y ((lambda (x) (+ x 1)) 2)))
```

Question: can lexical addresses be used to guide this substitution?

Come BACK!

(3)

Top level:



let does NOT bend the internal reference

end result: This local fact is not recursive.

For the same reason we cannot write

```
(let ((x 1)
      (y (+ x 1)))
  ....)
```

reference is to the containing env.

can expect the second x to refer to the first x

Use letx instead.

④ var-exp clause in lambda-calculus-subst

```
(and (element-of? (unparse-expression subst-id) free-vars-exp)
     (equal? (parse-expression id) subst-id))
```

Structure of program is recursive descent- it could be that the exp input was

(lambda (x) (+ x 1))

(say). So: when we get to computing the substitution of (say) y for all free occurrences of x in exp, this particular x should NOT be replaced. Even though - due to the recursive descent - the immediate context of x -

The body of the lambda-fun is just (+ x 1)

And x IS free in this body, even though it is not free in the overall expression.

(3) (lambda-calculus-subst E_1 E_2 x) is supposed to compute

$$E_1[E_2/x]$$

{ standard logical notation

"E_1 with all free occurrences of x replaced by E_2, without variable capture"

ie (lambda-calculus-subst exp subst-exp subst-id)
should return Λ properly
substituted

$$exp[subst\text{-}exp / subst\text{-}id]$$