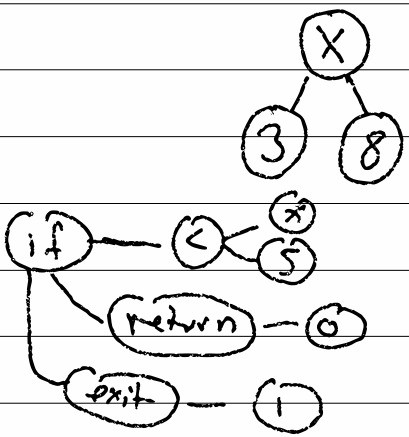
1-1/ 1 + 1 5 1+

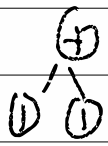
1 x 3 "3 x 8"



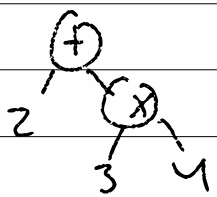

```
if (x < 5) {
    return 0; }
else {
    exit(1); }
```

(+ 1 1)

bp children



(+ 2 (x 3 4))

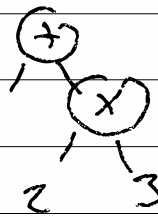


1-2) $J_0 \Rightarrow$ e := v | (+ e e)
v := number | (* e e)

(+ 1 (+ 2 3)) ∈ J_0

```
interface Joe { }
class JNumber implements Joe {
  int n;  JNumber(int _n) { n = _n; }}
class JPlus imp Joe {
  Joe left, right;  JPlus(...) }
class JMult imp Joe {
  Joe l, r;  JMult() ... {} }
```

(+ 1 (* 2 3)) → Sexpr =




```
new JPlus(
  new JNum(1),
  new JMult(
    new JNum(2))
    new JNum(3)))
```

= JP(JN(1), JM(JN(2), JN(3)))

```
class JPlus:
  def _init(l, r):
    this.l = l;
    this.r = r;
```

BST n := mt | (br num n n)

1-3/6 pp : J_0 => string

① pp n = itos (n)

② pp (+ e_L e_R) = "(" ++ pp(e_L) ++ "+" ++ pp(e_R) ++ ")"

③ pp (* e_L e_R) = "(" ++ pp(e_L) ++ "*" ++ pp(e_R) ++ ")"

① interface J0e { public String pp(); }

② class JNum { ...
    public String pp() {
        return intToStr(n); } }

③ class JPlus {
    public String pp() {
        return this.left.pp() ++ "+" + this.right.pp(); }

1-4/ big-step interpreter

```
interp : e -> v

interp n = n
interp (+ eL eR) = interp eL + interp eR
interp (* eL eR) = interp eL * interp eR
```

(1)

```
class JMult {
  public int interp() {
    return this.left.interp() * this.right.interp(); }}
```

```
(+ 1 2 3) = (+ 1 (+ 2 3))
```

desugar →

```
se = empty | (cons se se) | string
(a b c) = (pair "a" (pair "b" (pair "c" mt)))
(+ 1 2) = (p "+" (p "1" (p "2" mt)))
(+ 1 (+ 2 3)) = (p "+" (p "1"
                  (p (p "+" (p "2"
                            (p "3" mt)))
                     mt)))
```

1-5/ desugar for J0

("-" e) => (* -1 (desugar e))

("-" e1 e2) => (+ (d e1) (de ("-" e2)))

("+") => 0

("+" e1 more ...) =>

(+ (d e1) (d ("+" more ...)))

'*'

"x"

(x

= 1

"x"

se => J0 => v

desugar interp

compile bc => v

vm

2-11

desugarer

$(- \ e_1) \Rightarrow (\hat{*} \ -1 \ e_1')$

$(- \ e_1 \ e_2) \Rightarrow (\hat{+} \ e_1' \ (- \ e_2))$

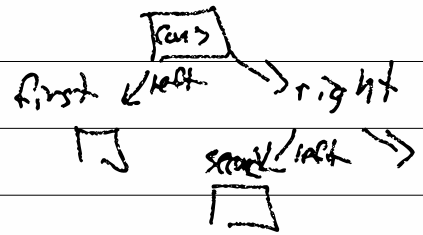
$(+) \Rightarrow 0$

$(+ \ e_1 \ e_2 \ \ldots) \Rightarrow (\hat{+} \ e_1' \ (+ \ e_2 \ \ldots))$

```
def
desugar(se):
    if isList(se) && length(se) = 2 &&
        first(se) == "-" then
```

```
> def length(se):
>     if isNull(se): return 0
>     else if cons(se): return 1 + length(rightTo)
>     else false
```

```
        new JMult(new JNum(-1), desugar(
                                  second(se)))
```




```
if isList(se) && len(se) = 3 && first(se) == "-"
then: return new JAdd(desugar(second(se)),
                      desugar(new cons("-",
                                       new cons(third(se), null))
```

22) cons(a, cons(b, cons(c, null))))

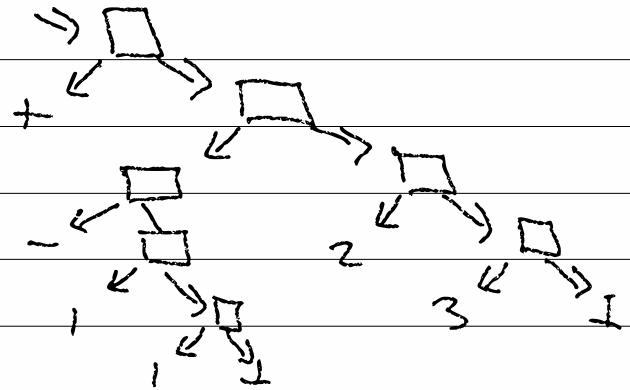
len = 3

(+ (- 1 1) len = 4

1 2 2 ← 3

3)

4



isList(se):

Null: True

cons: isList(right(se))

ow: False

Len(se):

Null: 0

cons: 1 + len(right(se))

2-3/ $J_0 \to J_1$

fun, args

$e ::= v \mid (e\ e\ \ldots) \mid (\text{if}\ e\ e\ e)$

c, t, f

$v ::= b$

$b ::=$ some set of constants

// in J_0, b = num | + | * | numbers | bools | prim

prim = +, -, *, /, ≤, <, =, >, ≥, ...

interp $v = v$

interp (if e_c e_t e_f) = interp e_k

where e_k = if interp e_c then e_t o.w. e_f

interp (e_f e_a ...) = $\delta(p, v_a \ldots)$

where p = interp e_f

v_a.. = interp e_a ..

$\delta : b \ldots \to b$

$\delta(+, 1, 2) = 3$ $\quad \delta(/, 1, 0) = \perp$

$\delta(<, 1, 3) = \text{true}$

24/ "small step interp" "big step"

e ⇒ e e ⇒ v

until its the same

interp Interp

```
Interp e =
  let e' = interp(e)
  if e == e' then
     ret e
  o.w.
     Interp (e')
```

(+ (+ 1 1) 2) ⇐ (+ (+ 1 1) (+ 1 1))

⇒ (+ 2 2)

⇒ 4

(+ 2 (+ 1 1))

```
int x = 1;
f ( x--, x++)      (1, 0)
                   (2, 1)
```

2-5/ step : e → e

```
step (if true  et ef) = et
step (if false et ef) = ef
step (p va ...) = δ(p, va ...)
step v = v
step (if e (&v) et ef) =
  (if (step e) et ef) or (if e (step et) ef)
step (vb ... e(&v) ea ...) =
  (vb ... (step e) ea ...)
```

A context

```
C ::= hole | if0 C e e
           | if1 e C e
           | if2 e e C
           | (e ... C e ...)
```

plug C e (C[e])

```
plug hole x = x
plug (if0 C e1 e2) x = if x e1 e2
plug (if1 e1 C e2) x = if e1 x e2
plug (e1 ... C e2 ..) x = (e1 ... x e2 ...)
```

2-6

step C [if true et ef] =
C [et]

step C [if false et ef] =
C [ef]

step C [p va ...] = C [δ(p, va ...)]

~~step~~ "parse" : e ⇒ C × e

step ⇒ e

interp e = if e ∈ v then e

C, e' = parse e e

e'' = step e'

plug C e''

redex

parse : e ⇒ C × e

parse (if ec et ef) =

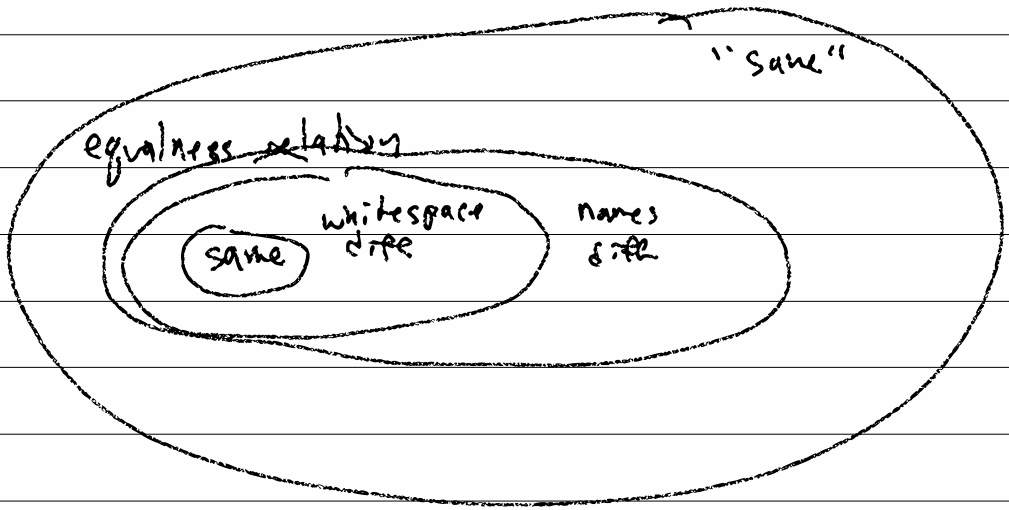
if ec ∈ v then (hole, e)

o.w. let C', e' = parse ec

(if0 C' et ef, e')

2-7 Answer: Contexts

Question: How do I know when two programs do the same thing?



$x = y$

$\forall x.\ fx = gx$

$\forall C.\ C[x] = C[y]$

C = hole x = y

C = (+ hole 2) x+2 = y+2

C = (map hole (list 1 2))

....

Observational Equivalence

2-8) $C ::= \text{hole} \mid \text{if } C\ e\ e \mid \text{if } e\ C\ e \mid \text{if } e\ e\ C \mid (e \ldots C\ e \ldots)$

$E ::= \text{hole} \mid \text{if } E\ e\ e \mid (v \ldots E\ e \ldots)$

"unique decomposition" ↓ unique E

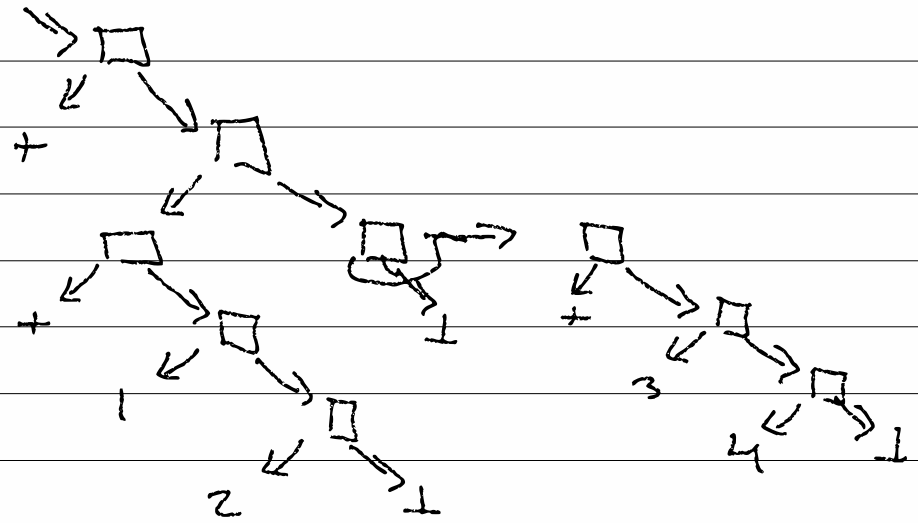
$\forall e.\ e \in v$ or $e = E[e']$ where $e' \notin v$

$\delta(1, 2) = 1 \qquad \delta(1, 1, 0) = 2$

$(+\ (+\ 1\ 2)\ (+\ 3\ 4))$ = tree = java tree

```
new JPlus (new JPlus (new JNum (1),
                      new JNum (2))
           new JPlus (new JNum (3),
                      new JNum (4)))
```

2-9



(+ (+ 1 2)
(+ 3 4))

3-1/ E = hole | (if E e e)
| (v ... E e ...)

step E[if true e+ ef] → E[e+]
step E[if false e+ ef] → E[ef]
step E[p va ...] → E[δ(p, va ...)]

interp e = case (parse e) of
false ⇒ e
(E, e) ⇒ e' = step e
E[e']

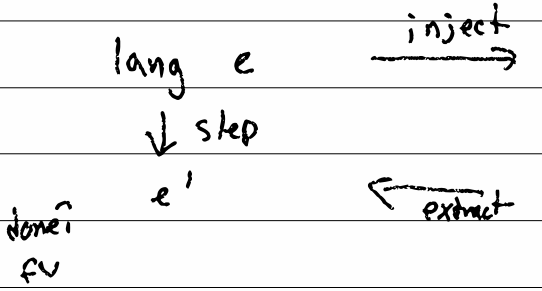
gigantic program [(+ (+ 1 1)
(+ 2 3))]

3-2/ Sp "language"

CC$_0$ "machine"

$e \to e$

$st \mapsto st$



$st = \langle e, E \rangle$ done? $\langle v, hole \rangle$

inject $e = \langle e, hole \rangle$

extract $\langle e, E \rangle = E[e]$

$\langle if\ e_c\ e_t\ e_f, E \rangle \mapsto \langle e_c, E[if\ hole\ e_t\ e_f] \rangle$

$\langle true, E[if\ hole\ e_t\ e_f] \rangle \mapsto \langle e_t, E \rangle$.

$\langle false, E[if\ hole\ e_t\ e_f] \rangle \mapsto \langle e_f, E \rangle$

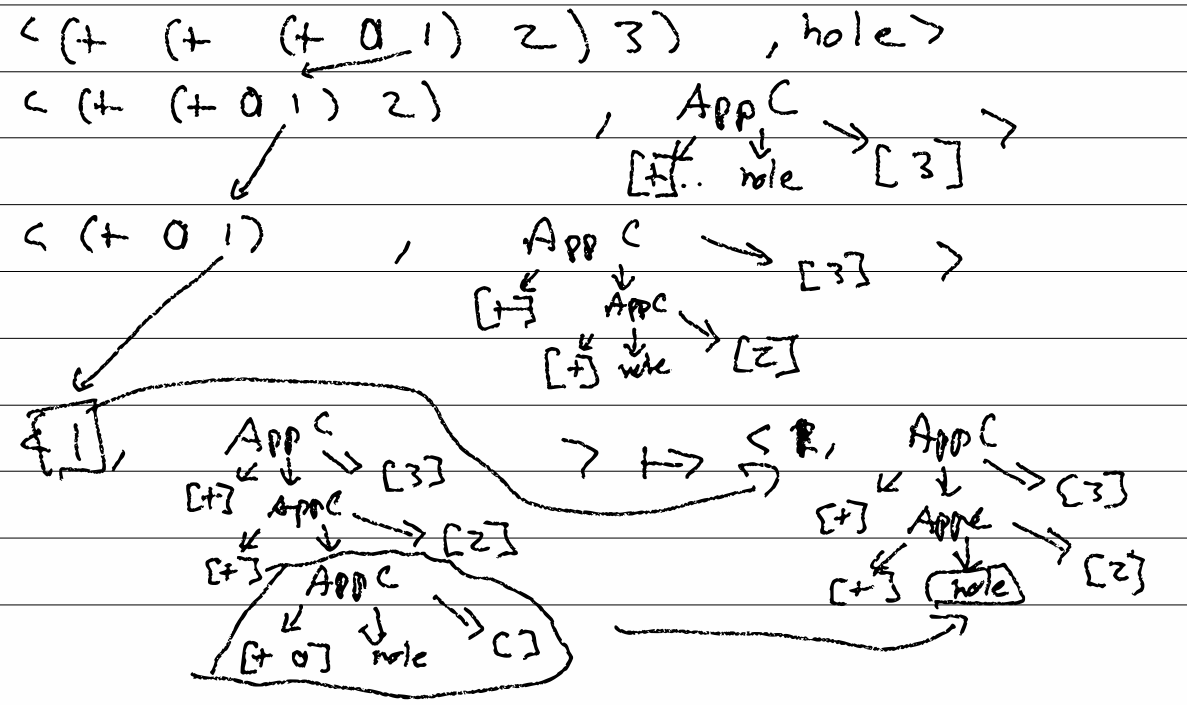
$\langle e_0\ e_1 \dots, E \rangle \mapsto \langle e_0, E[hole\ e_1 \dots] \rangle$

$\langle v, E[v_0 \dots hole\ e_1\ e_2 \dots] \rangle \mapsto \langle e_1, E[v_0 \dots v\ hole\ e_2 \dots] \rangle$

$\langle v_n, E[v_0 \dots hole] \rangle \mapsto \langle \delta(v_0 \dots v_n), E \rangle$

3) E = hole | if E e e | (v ... E e ...)

```
interface Context {
    Expr plug(Expr); }
Hole : Context {
    plug(e) = e; }
IfC : Context {
    Context c; Expr t, f;
    plug(e) = new If(c.plug(e), t, f); }
AppC : Context {
    List<V> vs; Context c; List<Expr> es;
    plug(e) = new App(vs ++ [c.plug(e)] ++ es); }
```



3-4// E = hole | if E e e | $v \ldots E\ e \ldots$

= top | if e e $\square$ | $(v \ldots)(e \sim) \square$

K = kret | kif e e K | kapp $\vec{v}$ $\vec{e}$ k

CK$_0$ machine $st = \langle e, K \rangle$

inject $e = \langle e, \text{kret} \rangle$

extract $\langle e, \text{kret} \rangle = e$

$\langle e, \text{kif } e_t\ e_f\ K \rangle$ = extract $\langle \text{if } e\ e_t\ e_f, K \rangle$

$\langle e, \text{kapp } (v \ldots)(e_1 \ldots) K \rangle$ = extract $\langle (v \ldots e\ e_1 \ldots), K \rangle$

done $\langle v, \text{kret} \rangle$

0 $\langle \text{if } e_c\ e_t\ e_f, K \rangle \mapsto \langle e_c, \text{kif } e_t\ e_f\ K \rangle$

1 $\langle \text{true}, \text{kif } e_t\ e_f\ K \rangle \mapsto \langle e_t, K \rangle$

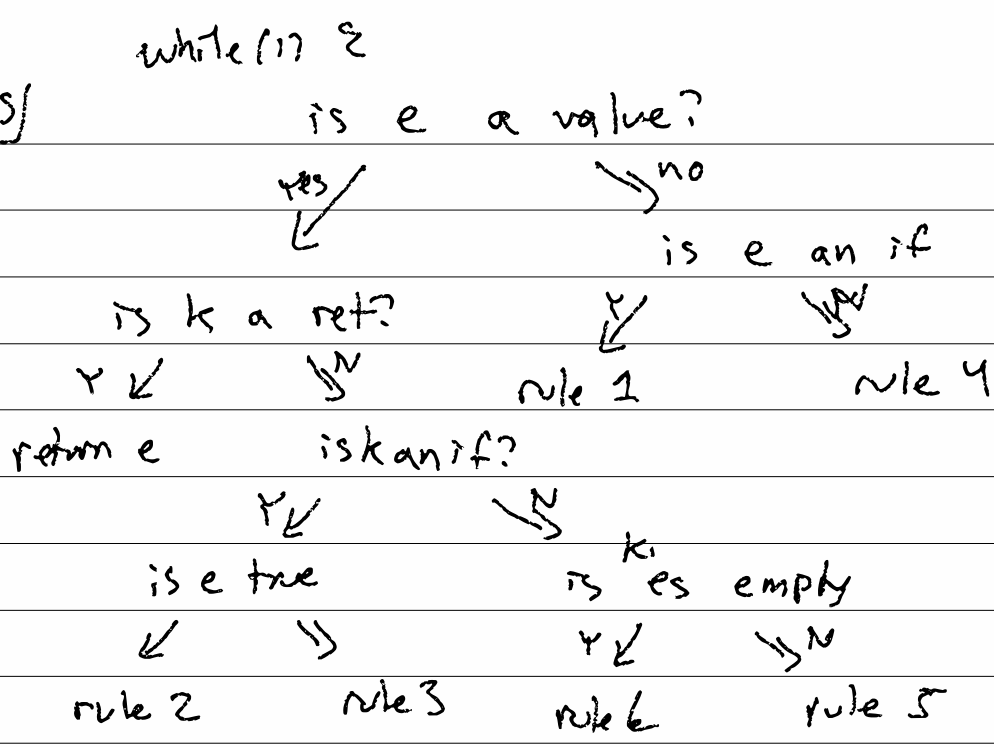
2 $\langle \text{false}, \text{kif } e_t\ e_f\ K \rangle \mapsto \langle e_f, K \rangle$

4 $\langle e_0\ e_1 \ldots, K \rangle \mapsto \langle e_0, \text{kapp } ()\ (e_1 \ldots)\ K \rangle$

5 $\langle v_1, \text{kapp } (v_0 \ldots)\ (e_0\ e_1 \ldots)\ K \rangle$
$\mapsto \langle e_0, \text{kapp } (v_0 \ldots v_1)\ (e_1 \ldots)\ K \rangle$

6 $\langle v_n, \text{kapp } (v_0 \ldots)\ ()\ K \rangle$
$\mapsto \langle \delta(v_0 \ldots v_n), K \rangle$

3-5)



rule 1:

k = new kif (e.true, e.false, k)

e = e.cond;

~~jump R.PC~~

k is a stack and the stack (of c) = kontinuation

continuation

3-6/

```
struct if {
  expr h;
  expr *c, *t, *f; }

struct num {
  expr h;
  int n; }

struct app {
  expr h;
  expr *f, *args; }

expr *make_if(expr *c, t, f){
  if *p = malloc(sizeof ...))
  p->h.tag = IF;
  p->c = c; ...
  return p; }
```

```
struct expr {
  enum tag; }

enum tag {
  IF, NUM, APP,
  BOOL, PRIM,
  KRET, KIF,
  KAPP, CONS, NIL};
```

(+ (+ 1 1) 7 2)

```
make_add ( make_add (make_num(1),
                     make_num(1)),
           make_num(2));
```

0201122207

4-1/ J_0 $e = n \mid (+\ e\ e) \mid (*\ e\ e)$

J_1 $p =$ unary- (neg), not (1)
+, *, between (2)

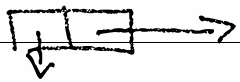
$e = n \mid p \mid (\text{if}\ e\ e\ e)$

$\boxed{(e \ldots)}$

class JApp impl JExpr {
List<JExpr> contents }

App
(+ 1 2) ⇒ List(+, 1, 2)
p n n

desugar
App([Prim(PLUS), Num(1), Num(2)])
(cons "+" (cons "1" (cons "2" null)))



(+ 1 (+ 2 3))
App([Prim(PLUS), Num(1), Num(2)
App([Prim(PLUS), Num(2), Num(3)])])

(+ 1 3 (+ 2 3))
δ(+ 1 3 5) = ⊥

4-2/ Expr * Delta (List<Expr*) args) {
if (len(args) == 3
&& args[0] == prim(PLUS)) {
ret new Num ((Num)args[1]->n + (Num)args[2]
->n);
}

(+) => 0

~~(+ n) => n~~

(+ n more ...) => (+ n (+ more ...))

desugar (cons "+" empty) = new Num(0);

delta (args)
args[0] (args)
⟶
args[0].apply (args[1...])

J_0 or J_1 : prog = e

4-3 / J_2 :

e := v | (e ...) | (if e e e)
| x

v := number | bool | prim | f

X ∈ some set of variable names

f ∈ some set of function names

prog := d ... e

d := (define (f x ...) e)

(define (Double x) (+ x x))

→ (Double (+ (Double 1) 3))

(define (Quad x) (Double (Double x)))

(Quad (+ 1 (Double 3)))

f(x) = 1 + x

f(3) ? = 1 + 3 = 4

f(x) = 1 − x

f(3+4) = 1 − (3 + 4)

= f(7)

= 1 − 7 = −6

~~= 2 + 4 = 2~~

Double (1 + 1) = Double (2)

(1+1) + (1+1) = 2 + 2

4-3 /

E = hole | if E e e

| (v ... E e ...)

~~$E[x] = \ldots$~~ $\Sigma / E[\text{if } T \text{ et } e_2] = E[e_1]$

$\Sigma / E[f \; v \ldots] = \ldots$

eval : $e \Rightarrow v$ .... smallstep $e \to e$

eval' : $p \Rightarrow v$

Σx ... smallstep $\Sigma \; e \Rightarrow e$

!

?

$\Sigma : f \to d$

eval' Σ (define (f x ...) e) : more

= eval' $\Sigma[f \mapsto d]$ more

eval' Σ e = do smallstep

$f(x) = 1 + x$

$f(4)$

$\Sigma / E[f \; v \ldots] = E[e[x \leftarrow v] \ldots]$

if $\Sigma(f) =$ (define (f x ...) e)

4-4 / $e[x \leftarrow v]$ is pronounced

e where xs are replaced with

v

subst x v e → e

subst x v v' = v'

subst x v x = v

subst x v x' = x'

subst x v (if e_c e_t e_e) =

(if $e_c[x \leftarrow v]$ $e_t[x \leftarrow v]$ $e_e[x \leftarrow v]$)

subst x v (e ...) =

(e[x ← v] ...)

```
interface JExpr {
    JExpr subst(Variable x, JExpr v);}
class JVar {
    subst(x, v) { if (x == this.x)
                    return v
                  return this; }}
class JIf {
    subst(x,v) {
    new JIf(this.c.subst(x,v), this.t.subst
                                  (x,v),
            this.f.subst(x,v)); }}
```

4-5

CK

6: $\langle v_n , kapp((\overset{\bullet}{v}_0 \ldots), (), k) \rangle$

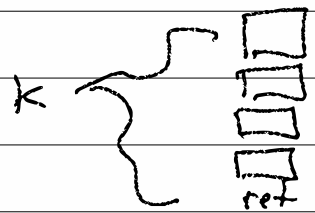
$\mapsto \langle \delta(v_0 \ldots v_n) , k \rangle$

7: $\langle v_n , kapp((f\ v_0 \ldots), (), k) \rangle$

$\mapsto \langle e[x_i \leftarrow v_i] , k \rangle$

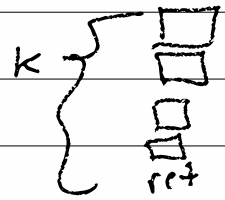
where $\Sigma(f) = (\text{define } (f\ x_0 \ldots x_n)\ e)$

$c = 2$ kapp (Expt 7)



$\Rightarrow$

$c = \ldots 7 \ldots 2 \ldots$



$\langle (e_0\ e_1 \ldots) , k \rangle$

$\mapsto \langle e_0 , kapp((), (e_1 \ldots), k) \rangle$

4-6 (define (F x) (F x))
(F 10)

$\Rightarrow \Sigma = [F \mapsto (\text{define } (F\ x)\ (F\ x))]$

$e = (F\ 10)$

$k = k_{ret}$

$\langle F\ 10, k_{ret} \rangle$ ⟸

$\langle F, k_{app}((), (10), k_{ret}) \rangle$

$\langle 10, k_{app}((F), (), k_{ret}) \rangle$

$\Sigma(F) = (\text{define } (F\ x)\ (F\ x))$

$f \quad x\ldots \quad e$

$\langle e[x\ldots \leftarrow v\ldots], k_{ret} \rangle$

$(F\ x)[x \leftarrow 10]$

$\langle (F\ 10), k_{ret} \rangle$ ⟸

error ⟹ Stack trace

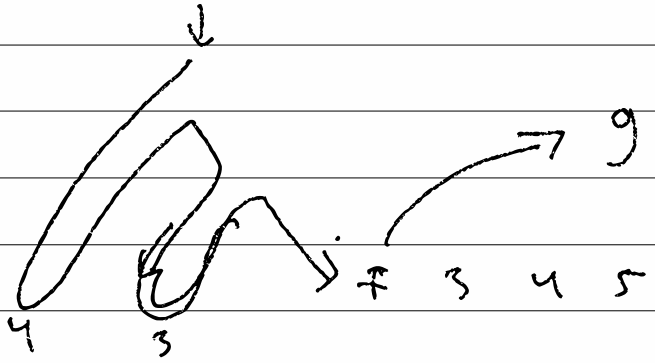
⇒ fun "at fault" — F

fun that called F — G

G — H

"past"

4-7)



$g(\cdot - \_\_)$

$x = f\ 3\ 4\ 5$

$h(x)$

6-1/ C = hole | if C e e
| if e C e
| if e e C
| e ... C e ...

E = hole | if E e e
| v ... E e ...

if true (+ 1 2) 4

E = hole e = ↑

C = hole e = ↑

C = if true hole 4
e = (+ 1 2)

(+ 1 2)

E = hole e = (+ 1 2)

find-redex B = E[e]

step e = e'

6-2) fr (if e t f) =

if (value? c)

(hole, e)

o.w. (E, e) = fr c

(if(E t f, e)

fr (app es) =

for e in es

if (value? e)

(+ 1 2)

$\langle e_0\ e \cdots, k \rangle$

$\mapsto \langle e_0, kapp((), (e \cdots), k) \rangle$

$\langle v_n, kapp((v_0 \cdots), (e_{n+1} \cdots), k) \rangle$

$\mapsto \langle e_1, kapp((v_0 \cdots v_n), (e_{n+1} \cdots), k) \rangle$

$\langle v_n, kapp((p\ v_0 \cdots), (), k) \rangle$

$\mapsto \langle \delta(p, v_0 \cdots v_n), k \rangle$

6-3/ J_2 = PASCAL or C
top-level functions
CK = have the map $f \to d$
and we have subst

$\langle v_n, kapp((f\ v_0 \ldots), (), k) \rangle$
$\longmapsto \langle e[x_0 \leftarrow v_0] \ldots [x_n \leftarrow v_n], k \rangle$
where $\Sigma(f) =$ define $(f\ x_0 \ldots x_n)\ e$

$x[x \leftarrow v] = v$
$y[x \leftarrow v] = y$
$u[x \leftarrow v] = u$
$(\text{if } c\ t\ f)[x \leftarrow v] = (\text{if } c[x \leftarrow v]\ t[x \leftarrow v]\ f[x \leftarrow v])$
$(e \ldots)[x \leftarrow v] = (e[x \leftarrow v] \ldots)$

$\langle \text{if } c\ t\ f, k \rangle$
$\longmapsto \langle c, kif(t, f, k) \rangle$

OLD: NO rule for a variable in the CPS
$\langle x, k \rangle \longmapsto \ldots$

NEW:
$\langle x, k \rangle \longmapsto$ finally do the subst

G-Y/ CEK $st = \langle e, env, k \rangle$

$env = \emptyset \mid env[x \leftarrow v]$

$k = khalt \mid kif\ e\ e\ k$

$\mid kapp\ \vec{v}\ \vec{e}\ k$

$\langle x, env, k \rangle \mapsto \langle env(x), \emptyset, k \rangle$

$\langle if\ c\ t\ f, env, k \rangle$

$\mapsto \langle c, env, kif\ t\ f\ k \rangle$

$\langle true, env, kif\ t\ f\ k \rangle$

$\mapsto \langle t, env, k \rangle$

$\langle e_0\ e_1 \ldots, env, k \rangle$

$\mapsto \langle e_0, env, kapp\ ()\ (e_1 \ldots)\ k \rangle$

$\langle v_1, kapp\ (v_0 \ldots)\ (e_0\ e_1 \ldots)\ k \rangle$

$\mapsto \langle e_0, env, kapp\ (v_0 \ldots v_1)\ (e_1 \ldots)\ k \rangle$

$\langle v_n, env, kapp\ (v_0 \ldots)\ ()\ k \rangle$

$\mapsto \langle e, \emptyset\ [x_0 \leftarrow v_0] \ldots [x_n \leftarrow v_n], k \rangle$

where $\Sigma(f) = define\ (f\ x_0 \ldots x_n)\ e$

6-5/

```
define f x = x + z
define g z = f 4
g 2
```

```
define f x = 3
define g z = (f 1) + x
g 2
```

6-6

CEK = < e, env, k >

env = ∅ | env [x ← v]

k = kret | kif env t f k

| kapp $\vec{v}$ env $\vec{e}$ k

< x, env, k > ↦ < env(x), ∅, k >

< if c t f, env, k > ↦ < c, env, kif env t f k >

< true, _, kif (env, t, f, k > ↦ < t, env, k >

< e0 e1 ..., env, k >

↦ < e0, env, kapp ((), env, (e1 ..), k)>

< vn, _, kapp ((v0 ...) , env, (e0 e1 ..), k)>

↦ < e0, env, kapp ((v0 .. vn), env, (e1 ...) k)>

< vn, _, kapp ((f v0 ...), _, (), k)>

↦ < e, ∅ [x0 ← v0] ... [xn ← vn], k >

where Σ(f) = define (f x0 ... xn) e

"dynamic scope" — equal

- emacs lisp
- JS / Py / Ruby / perl / PHP / etc

specific vars are always dynamic

new = ∅ [A ← env(A)] [B ← env(B)]

"this"

2-7 PASCAL/C – all funs are top-level

p = d ... e

JS = (x) => 1 + x

Py = lambda : x : 1 + x

C++ = [](int x){return 1+x;}

J_3

e = v | e e ... | if e e e | x

v = b | (λ (x ...) e) — new

b = num | bools | prim // No f's

E = hole | if E e e | v .. E e ...

E [(λ (x0 .. xn) e) v0 .. vn] =
E [e [x0 ← v0] .. [xn ← vn]]

((λx.
((λy.
(x + y) 7)) 8)) => 15

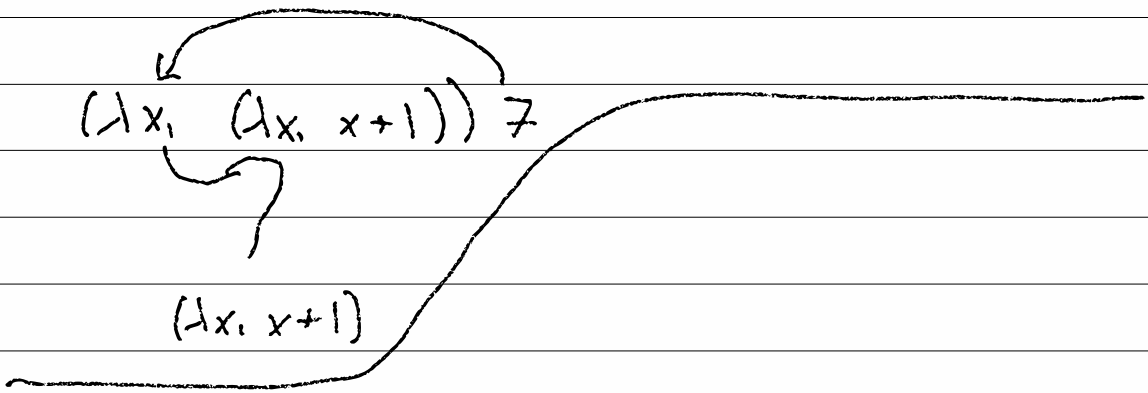
let x = 8 in
let y = 7 in
x + y

let x = e1 in e2 => (λ x. e2) e1

let x = 8 in
let x = x + 1 in
x + x

6-8/

$$(\lambda\ (x_0 \ldots x_n)\ e)\,[y \leftarrow v]$$
$$= (\lambda\ (x_0 \ldots x_n)\ e\,[y \leftarrow v])$$
unless $y \in x_0 \ldots x_n$



OLP
machine v = theory v

new
$(v := b \mid \bigstar) \neq (v := b \mid \lambda(x \ldots)\ e)$
closure $(\lambda(x \ldots)\ e,\ env)$

$\langle \lambda(x \ldots).\ e,\ env,\ k \rangle \longmapsto \langle clo(\lambda(x \ldots) e,\ env),\ \emptyset,\ k \rangle$
$\langle v_n,\ -,\ kapp((clo\,\lambda(x_0 \ldots x_n).\ e,\ env)\ v_0 \ldots),\ \rightarrow$
$(),\ k) \rangle$
$\longmapsto \langle e,\ env\,[x_0 \leftarrow v_0] \ldots [x_n \leftarrow v_n],\ k \rangle$

7-1) J_3 : $v ::= \ldots \mid \lambda(x \ldots)\ e$

let $x = e_1$ in e_2

$\Rightarrow (\lambda(x)\ e_2)\ e_1$

let x = 1 ; in
let y = 2 ; in
x + y

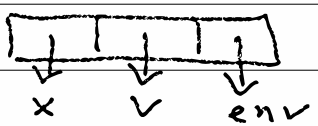
$(\lambda(x)$
$(\lambda(y)$
$(+\ x\ y)\ 2)\ 1$

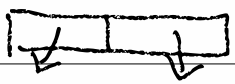
CK_1 : $v ::= \ldots$
~~$\mid \lambda(x)\ e$~~
$\mid clo(\lambda(x \ldots)\ e,\ env)$

$(\lambda(y)\ (+\ 1\ y))\ 2$

$\langle \lambda(x \ldots)\ e,\ env,\ k \rangle$
$\mapsto \langle clo(\lambda(x \ldots)\ e,\ env),\ \emptyset,\ k \rangle$

$\langle v_n, \_, kapp((c\ v_0 \ldots), \_, (), k) \rangle$
where $c = clo(\lambda(x \ldots) e,\ env)$
$\mapsto \langle e,\ env[x_0 \mapsto v_0] \ldots [x_n \mapsto v_n],\ k \rangle$

$env = \bot \mid$ 

$clo =$ 

7-2)

```
let y = 3 ; in
let z = 8 ; in
λ(x) (+ x y)
```

clo ⇒ [8, 3, 19, 27, 36]

z ⇒ 8
y ⇒ 3
x ⇒ 19
a ⇒ 27
b ⇒ 36
...

$(\lambda.\ (+\ \hat{0}\ \hat{1}),\ [\hat{1}])$

static address

SA = nat env = vector v

FLAT-CLOSURES [⌣ , 3]

SA = (nat, nat) env = ↓ , vector v

$(\vec{0},0)$ $(\hat{1},1)$ [8, 3, 19, 27, 36] env

↑ , [⌣ , ⌣] NESTED CLOSURES

7-3/

$= v \mid (\text{if } e\ e\ e) \mid x \mid (e\ e) \mid (p\ e\ e)$

$v = b \mid \lambda (x) e$

$b = num \mid bools \mid prim$

$prim = + \mid - \mid * \mid \div \mid <$

$(+\ 1\ 2) \quad | \ (\lambda\ (x\ y)\ (+\ x\ y))$

$1 \quad 2 \ )$

$(((\lambda x.\ \lambda y.\ (+\ x\ y))\ 1)\ 2)$

$= (+\ 1\ 2)$

$e = v \mid x \mid e\ e \qquad E = hole$

$v = \lambda x.\ e \qquad \mid (E\ e)$

$\mid (v\ E)$

$E[(\lambda x.e)\ v] \mapsto E[e[x \leftarrow v]]$

7-4/ What is a Bool really, man?

if True A B = A

if False A B = B

True = $\lambda x.\lambda y.\ x$

False = $\lambda x.\lambda y.\ y$

if = $\lambda c.\lambda x.\lambda y.\ c\ x\ y = \lambda c.\ c$

if True A B = True A B = A

NOT T = F

NOT F = T

NOT = $\lambda b.\lambda x.\lambda y.\ b\ y\ x$

```
interface Bool { int choose(int, int); }
class True : Bool { int choose(x, y) = x }      // True()
class False : Bool { int choose(x, y) = y }     // False()
class Not : Bool { Not(Bool b) { this.b = b; }  // Bool b;
                   int choose(x, y) {
                     return b.choose(y, x); } }
```

7-5/ What is a number?

zero := doesn't do something

one := does something once

two := does it twice

add x^n y^m = ~~[illegible]~~ $n+m$

zero := $\lambda f.\ \lambda x.\ x$

one := $\lambda f.\ \lambda x.\ f\ x$

two := $\lambda f.\ \lambda x.\ f\ (f\ x)$

add1 := $\lambda n.\ \lambda f.\ \lambda x.\ f\ (n\ f\ x)$

add := $\lambda n.\ \lambda m.\ \lambda f.\ \lambda x.\ n\ f\ (m\ f\ x)$

zero? := $\lambda n.\ n\ (\lambda x.\text{FALSE})\ \text{TRUE}$

mult := $\lambda n.\ \lambda m.\ \lambda f.\ \lambda x.\ n\ (m\ f)\ x$

(two, two)

two (two f) x

$(\lambda x.\ f\ f\ x)$

$(\lambda x.\ f\ f\ x)\ ((\lambda x.\ f\ f\ x)\ x)$

$f\ f\ f\ f\ x$

7-6/ Pair

fst (pair A B) = A

snd (pair A B) = B

pair = λa.λb.λc. if c a b

fst = λp. p TRUE

snd = λp. p FALSE

subl := λn. fst (n (λp. pair (snd p) (add1 (snd p))) (pair z z))

mkfac := λfac. λn.
  if (zero? n)
    1=one
    (addlt n (fac (subl n)))

g(x) = x ∪ {a,b} f(x) = 17 · x

fac := mkfac fac x = F x

↑ x ↑ F ↑ x

7-7/ Fixed point of a lambda?

FIX F = x F x = x

F (FIX F) = FIX F

= Z-combinator

FIX := λF.((λx. F (λv. x x v))
(λx. F (λv. x x v)))

FIX F := ((λx. F (λv. x x v))
(λx. F (λv. x x v))) = A

= F (λv. A A v)

= F (A A)

= F ((λx. F(λv. x x v))
(λx. F(λv. x x v)))

λx.λy. x y (x)

= F (FIX F)

Lambda-Calculus n add 1 ⇒ λx. x+1

Church Numeral / Church encoding

Talk about CCO

8-1 Lambda - Calculus

---

tiny : e → e

tiny (if T et ef) = et

tiny (if F et ef) = ef

tiny (p $v_0 \cdots v_n$) = $\delta(p, v_n)$

small : e → e

small e = let C, e' = find-redex e

big : e → v

let e'' = tiny e'

let C [e'']

big e =

if e ∉ V : return e

o.w. big (small e)

CCO : C × e → C × e

CCO C e =

C' × e' = move-context C e

e'' = tiny e'

ret (C', e'')

CCO* C e = if e ∉ V and C = hole, ret C e

o.w. CCO* (CCO C e)

big' : e → e

extract (CCO* (inject e))

inject = (hole, e) extract (hole, v) = v

8-2)

$\langle$ if e_c e_t e_f, E $\rangle$

$\mapsto$ $\langle$ e_c, E[if hole e_t e_f] $\rangle$

A

(+ 1

B

C D F

(if (zero? 1) (+ 2 3) (+ 4 5))

inject A = $\langle$ A, hole $\rangle$

ccO $\langle$ A, hole $\rangle$

$\mapsto$ $\langle$ + , hole[(hole 1 B)] $\rangle$

= $\langle$ + , (hole 1 B) $\rangle$

$\mapsto$ $\langle$ 1 , (+ hole B) $\rangle$

$\mapsto$ $\langle$ B, (+ 1 hole) $\rangle$ = E[if hole e_t e_f]

$\mapsto$ $\langle$ C, (+ 1 (if hole D F)) $\rangle$

$\mapsto\mapsto$ $\langle$ False, = 7

$\mapsto$ $\langle$ B, (+ 1 hole) $\rangle$

$\mapsto\mapsto\mapsto$ $\langle$ 9, - 7

$\mapsto$ $\langle$ 10, hole $\rangle$

$\rightarrow$ extract $\rightarrow$ 10

8-3/ Lambda-calculus

$$e = x \mid e\ e \mid \lambda x.e$$

J_3 doesn't recursion (except via Z)

$J_3 \Rightarrow J_4$

$v = \ldots \mid \lambda (x \ldots) e \mid \lambda * (x \ldots) e$

a.map (lambda x: x+1)

lambda fib (n): ...

rec args

| let fib = λn. .... (fib (subl n)) (fib: unbound)

| let x = xe in be
:= (λx. be) xe

let fib = λ inner_fib : n. ....
inner_fib (subl n)

"(define (f x...) xe) ; b"
⇒ "let f = λf (x...). xe in b"

8-4/

$$E[(l\ v_0 \cdots v_n)] = E[b[f \leftarrow l][x_0 \leftarrow v_0] \cdots]$$

where $l = (\lambda\ f\ (x_0 \cdots x_n)\ b)$

$$\langle \lambda f\ (x \ldots)\ b\ ,\ env\ ,\ k \rangle \mapsto \langle c\ ,\ \emptyset\ ,\ k \rangle$$

where $c = clo(\lambda f\ (x \ldots)\ b,\ env')$

$env' = env[f \leftarrow c]$

```
switch (tag(c)) {
  case LAMBDA:
    envp = make_env(env, c->fun, NULL);
    c = make_clo( c , envp )
    envp -> val = c;
    env = NULL;
    break;
```

8-5/

```
while(1) {
int x, y;
scanf("%d", &x);
scanf("%d", &y);
*x = *y; }
```

deref (malloc (4), 5) = 1

Algebraic data types

dt ::=	0	—	____
|	1	—	void
|	dt + dt	—	interface invariants
|	dt x dt	—	pair

type	construct	destruct
1	void	—
0	—	—
dt x dt	pair	fst, snd
dt + dt	left, right	case / switch / if

case (left a) X Y => X a

case (right a) X Y => Y a

8-b)

$$\text{Bool} = 1 + 1$$

$$\text{Nat} = 1 + \text{Nat}$$

$$\text{Bin} = 1 + \text{Bin} + \text{Bin}$$

$$\text{List}(A) = 1 + (A \times \text{List}(A))$$

$$\text{BinT}(A) = 1 + (A \times \text{BinT}(A) \times \text{BinT}(A))$$

$$\text{BinT}'(A) = A + (\text{BinT}'(A), \text{BinT}'(A))$$

$$SE = 1 + \text{Atom} + (SE, SE)$$

$$d_A\, 0 = 0$$

$$d_A\, 1 = 0$$

$$d_A\, A = 1$$

$$d_A\, B = 0$$

$$d_A\, X + Y = d_A X + d_A Y$$

$$d_A\, X * Y = d_A X * Y + X * d_A Y$$

$$d_A\, \text{List}(A) = \text{Zipper}(A)$$

9-1/ (+ 1 2)

JAPP(+, 1, 2).asc()

prim 1 2

= "make_japp(make_jprim(PLUS), ...)"

write_to_file("x.c", 0.asc())

$J_4 \rightarrow J_5$

e := x | v | (e e ...) | (if e e e)

case e as (inl x) ⇒ e or (inr x) ⇒ e

v := num | bool | prim | λ x (x ...) e

unit | pair v v | inl v | inr v

prim := .... | pair | inl | inr

| fst | snd

$E[\text{fst (pair } v_1\ v_2)] = E[v_1]$ $E[\text{snd (pair } v_1\ v_2)] = E[v_2]$

$E[\text{case (inl } v) \text{ as (inl } x_1) \Rightarrow e_1 \text{ or (inr } x_r) \Rightarrow e_r] \Rightarrow E[e_1[x_1 \leftarrow v]]$

$E[\text{(inr } v)] \Rightarrow E[e_r[x_r \leftarrow v]]$

9-3/// List is either empty
or a cons with a thing
and another list

```
empty := inl unit
cons := λ(data rest). inr (pair data rest)

length := λrec (l).
  case l of
    (inl _) -> 0
    (inr p) -> 1 + rec (snd p)

map := λrec (f l).
  case l of
    inl _ -> l
    inr p -> cons (f (fst p))
                  (rec f (snd p))

reduce := λrec (f z l).
  case l of inl _ -> z
            inr p -> rec f (f z (fst p))
                           (snd p)
```

9-3/ Reduce (+) 0 (cons 1 (cons 2 (cons 3 empty)))

= reduce (+) 1 (cons 2 (cons 3 mt))

= reduce (+) 3 (cons 3 mt)

= reduce (+) 6 mt

= 6

true := inl unit

false := inr unit

if e_c e_t e_f := case e_c of inl _ => e_t
inr _ => e_f

inl _ inr inl _ inr inr _

pair => tuple fst/snd => π/.proj fst = π_0 snd = π_1

case$^{(2)}$ => case (2) inl/inr => choice I

```
obj_t* delta_pair (obj_t* l, obj_t* r){
    ret make_pair (l, r); }
obj_t* delta_fst (obj_t* o) {
    ret ((pair_t*) o) -> fst; }
```

9-4/ $e ::= obj\ \{ x : e, \ldots \}$

$\mid\ e.x$

$v ::= obj\ \{ x = v \ldots \}$

$E[\ obj\ \{ x_0 : v_0 \ldots x_i : v_i \ldots x_n : v_n \}\ .\ x_i\ ] \Rightarrow E[\ v_i\ ]$

$\{\}$

empty % = empty

set o x e = (cons (pair "x" e) o) (added)

{ x = e }

e.x = lookup "x" e

lookup := λrec (field obj).

case obj of inl _ ⇒ (rec field obj)

inr p ⇒ if string=? field (fst p)

(snd p)

(rec field (snd p))

CEk K ::= Kcase (x1, e1, xr, er, env, k) Kcase(c, k)

< case es of inl x1 ⇒ e1 or inr xr ⇒ er, env, k >

↦ < es, env, Kcase (x1, e1, xr, er, env, k) >

< inl v, _, Kcase (x1, e1, xr, er, env, k) >

↦ < e1, env[x ← v], k >

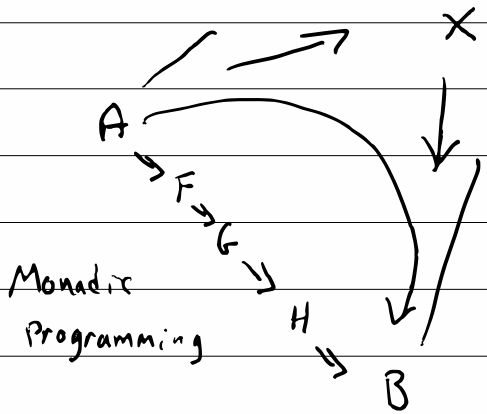
9-5 Mutation

$A \to B = \{(a,b), \ldots, (a,b)\}$

JS	Math
`const x = f(3);` (9(f))	$x = f(3)$
`console.log(x);` "42"	
`const y = f(3);` y=x	$y = f(3)$
`console.log(y);` "??"	
	$x = y$? ✓
`function f(x) {`	$f(3) = f(3)$
`return x + 39; }`	

`let c = 0; const f = (x) => x + 39 + c++;`

```
if p f()
...
if g c()
...
f(3)
```



12-1

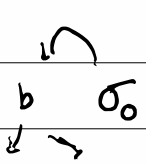
Goal: add mutation

e ::= .... | box e | unbox e | set-box! e e

E = .. | box E | unbox E | set-box E e | set-box v E

```
struct box { void *p; }
box (int x) { ip = malloc(int)
              box b = { p = ip }
              *ip = x;
              ret b; }
```

```
let b = box 6 ; in
(set-box! b 8;
 unbox b) + (unbox b)
```



(set-box! (box 6) 8; unbox (box 6)) + (unbox (box 6))

$\emptyset$ [ ] (set-box! σ_0 8; unbox σ_0) + (unbox σ_0)

$\Rightarrow \emptyset[\sigma_0 \mapsto 8]$ (unbox σ_0 + unb σ_0) $\Rightarrow \sigma[\sigma_0 \mapsto 8]$ / (8 + 8) = 16

12-2/ small_step : $e \to e$

$\Sigma \times e \to \Sigma \times e$

$(M, S, pc) \to (M', S', pc')$

Σ = store (memory / heap)

ptrs $\Rightarrow$ vals

$\sigma \Rightarrow v$

inject $e = \emptyset / e$

extract $\Sigma / v = v$

$v ::= \ldots \mid \sigma$

$\Sigma / E[\text{if } T \text{ et ef}] \to \Sigma / E[\text{et}]$

$\Sigma / E[\text{box } v] \to \Sigma[\sigma \mapsto v] / E[\sigma]$

where σ d.n.o.i. Σ (σ = malloc)

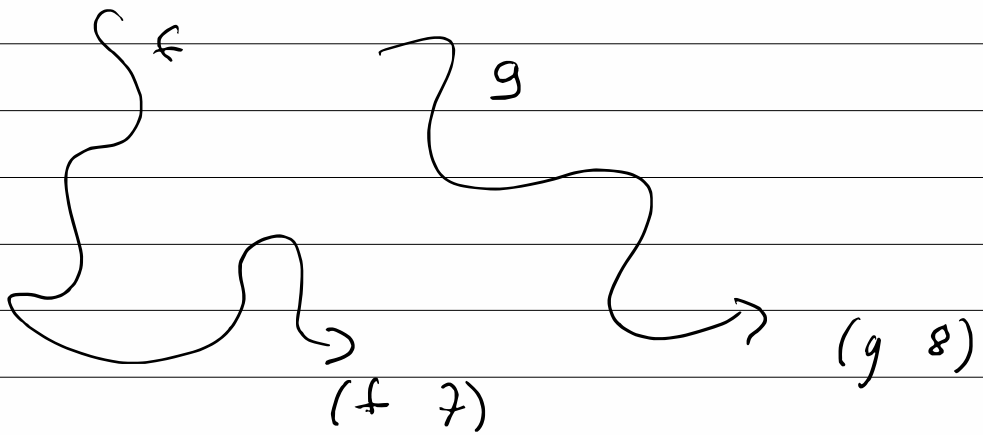
$\Sigma / E[\text{ubobx } \sigma] \to \Sigma / E[\Sigma(\sigma)]$

$\Sigma / E[\text{set-box! } \sigma\ v] \to \Sigma[\sigma \to v] / E[v]$ (C)

$E[\underset{\text{void}}{\text{unit}}]$

12-3/

```
let b = box 0 in
let f = λx. set-box! b (x+1);
            x * 2 in
let g = λy. y + unbox b in
```



CEK$_3$

CESK$_0$
+
store

CEK$_4$

v = ... | ptr n
p = ... | box
| unbox
| set-box

$\langle$ v, env, sto, kapp (clo(λx.e, env'), k$\rangle$
$\mapsto$ $\langle$ e, env'[x $\to$ v], sto, k$\rangle$

$\langle$ if e_c et cf, env, sto, k$\rangle$
$\mapsto$ $\langle$ e_c, env, sto, kif(env, et, ef, k)$\rangle$

$\langle$ v, env, sto, kbox(k)$\rangle$
$\mapsto$ $\langle$ σ, env, sto[$\sigma \mapsto$ v], k$\rangle$
where σ = malloc(sto)

$\langle$ box e, env, sto, k$\rangle$
$\mapsto$ $\langle$ e, env, sto, kbox(k)$\rangle$

12-4/ v = ... | pair v v

| box σ

Question: should we add

set-fst : pair A B × A → ()

Set-snd : pair A B × B → ()

```
mpair a b = pair (box a) (box b)
mpair-set-fst p a' = setbox (fst p) a'
mpair-fst p = unbox (fst p)
```

```
let f xi =
    let x = box xi in
    set! x 5;
    x
in
let v = (box 7) in
set! v 8
v + v
```

e ::= .... | set! x e

(1) (2)

12-5/ Always store vars as pointers

OLD:

$$\Sigma \,/\, E[(\lambda x.\, e)\ v] \to \Sigma \,/\, E[e[x \leftarrow v]]$$

NEW:

$$\Sigma \,/\, E[(\lambda x.\, e)\ v] \to \Sigma[\sigma \mapsto v] \,/\, E[e[x \leftarrow \text{unbox } \sigma]]$$

where $\sigma = \text{malloc}(\Sigma)$

$$\Sigma \,/\, E[\text{set!}\ (\text{unbox } \sigma)\ v] \to \Sigma[\sigma \mapsto v] \,/\, E[v]$$

desugar $(e_1 ; e_2)$ =

let -nggmncihannim = e_1 in

e_2

$= (\lambda \_ .\, e_2)\ e_1$

12-b/ Fancy desugar

desugar $(\lambda x.$ ... x ... set! x e ... $)$

$= \lambda x_i.$ let x = box x_i in ... (unbox x) ... set-box! x e ...

desugar M $(\lambda x.\, e) =$

if modified x e then $\lambda x_i.$ let x = box x_i in desugar $(M \setminus \{x\})$ e

o.w. $\lambda x.$ desugar M e

desugar M x =

if $x \in M$ then unbox x

o.w. x

12-7

eval e

= stdlib

eval (let ..... in

e)

→ prints 11

→ compile

→ runs

desugar (map) = \f. . . . .)

15-1/ 1 / 0 — δ is undefined (partial)

(5 1) — fun app needs a λ or p

set-box! 7 2 — set-box needs a box

$x \to y \to z \to (\div\ 1\ 0) \Rightarrow$

eval (p) = v iff $p \to^* v$

(eval is partial)

$v = \ldots \mid bad\ bad\ bad$

$E[(v_0\ v \ldots)] \to bad\ bad\ bad$

where $v_0 \notin p, \notin \lambda \ldots$

$v = \ldots \mid err_1 \mid \ldots \mid err_{255} \mid ok$

$E[(p\ v \ldots)] \to err_{17}$

$\delta(p, \vec{v}) = \lightning$ $\qquad E[a] \to E[b]$

$e = \ldots \mid abort\ e \qquad E' = E$

$E[abort\ e] \to e$

$\langle abort\ e, env, k \rangle \longmapsto \langle e, env, k_{ret} \rangle$

Jq:

15-2/ e = ... | throw e
| try e with catch e

try (+ 1 (throw 2))
with catch (λx. (- x 1)) => 1

E = ... | try E with catch e
| try E with catch v

E[try v with catch u] → E[v]
E[try L[throw v] with catch u]
→ E[u v]

L = E − (try E with catch v)

try (+ 1
(try (+ 2 (throw 3))
with catch (λ(v) (* v 2))))
with catch (λ(x) (* x 3)) => 7

15-3/ $K = \ldots \mid \text{preTry}\ K\ e\ env\ K$

$\mid$ ~~preTry~~ Try $K\ v\ K$

$\langle$ try e_b with catch e_h , env , $K\rangle$

$\mapsto \langle e_h, env, \text{preTry}\ K\ e_b\ env\ K\rangle$

$\langle v_h, \_, \text{preTry}\ K\ e_b\ env\ K\rangle$

$\mapsto \langle e_b, env, \text{Try}\ K\ v_h\ K\rangle$

$\langle v_{ans}, \_, \text{Try}\ K\ v_h\ K\rangle$

$\mapsto \langle v_{ans}, \_, K\rangle$

$\langle$ throw e, env, Try K v_h $K\rangle$

$\mapsto \langle v_h\ e, env, K\rangle$

$\mapsto \langle e, env, \text{kapp}\ (v_h)\ \_\ ()\ K\rangle$

$\langle$ throw e, env, kapp $(v \ldots)$ env' $(e \ldots)$ $K\rangle$

$\mapsto \langle$ throw e, env, $K\rangle$

preTry K e env' K

$\langle$ throw e, env, kret $\rangle \mapsto \langle e, env, \text{kret}\rangle$

15-4/ (f a b c) →

```
(let ([av a])
 (if (function? av)
   (if (= (function-arity av) 2)
     (unsafe-apply av b c)
     (throw "wrong num args"))
   (throw "not fun")))
```

```
(+ 2 (throw 0))
```

```
try ..... [(throw e)]
with catch
     (λ (x tryagain)
       (tryagain 8))
```

E[try L[throw e] with catch u]
→ E[u e
(λ (x) try L[x] with catch u)]

15-5/ First-class continuations

$e = \ldots \mid \text{callcc}\ e$

$E = \ldots \mid \text{callcc}\ E$

$E[\text{callcc}\ v] \to E[v\ (\lambda(x)\ \text{abort}\ E[x])]$

cek $v = \ldots \mid \text{kont}\ k$

$k = \ldots \mid \text{kcallcc}\ k$

$\langle v, \_, \text{kcallcc}\ k \rangle$

$\mapsto \langle v\ (\overset{\text{(kont)}}{k}), \_, k \rangle$

$\langle \text{callcc}\ e, env, k \rangle \mapsto \langle e, env, \text{kcallcc}\ k \rangle$

$\langle \text{if}\ c\ t\ f, env, k \rangle \mapsto \langle c, env, \text{kif}\ env\ t\ f\ k \rangle$

$\langle v, \_, \text{kapp}\ (\text{kont}\ k)\ \_\ () \to$

$\mapsto \langle v, \_, k \rangle$

15-6/

```
f= (λ (x)
    (callcc (λ (return)
      (if (zero? x)
          (return 2)) ;
      (print x)
      (/ 2 x))))
```

```
int f(int x){
  if (x == 0)
    return 2;
  printf("%d\n", x);
  return 2/x;}
```

```
(+ 1 (f 7))
(+ 1 (f 0))
```

return = λx. abort (+ 1 x)

```
(λ (x ...) b) => (λ (x ...)
                   (callCC (λ (return)
                     b )))
```

15-7

```
(define last-handler
  (box (λ (x) (abort x))))
(define throw
  (λ (v) ((unbox last-handler) v)))
desugar (try e1 with catch e2)
  = (try-catch* (λ () e1) e2)

try-catch* := (λ (Body new-handler)
  (let ([old-handler (unbox last-handler)])
    (call/cc (λ (here)
      (set-box! last-handler (λ (x) (set! lh oh)
                                    (here (nh x))))
      (let ([ans (body)])
        (set-box! lh oh)
        ans))))))
```

17-1/ €10) (5 3)

unsafe — just do something

.... (f x)

((€lo *) f) → code_ptr (x)

↓

jump (f+8)

(define (f x) (f (scanf))
(x 13))

unsafe = the language doesn't protect its abstractions

safe = DOES

C no abs, safe

€, C++ *(scanf()) (int*) o [2]

Java

JS

Py

Racket

} intend to be safe but loopholes

17-2

```
(define L                          ⇒ dlload
  (load "libOpenGL"))
(define glDraw                     ⇒ dlsym
  (extract L "glDraw"))
(glDraw .....)
```

desugar `(define (f x ...) body) ; more`

⇒

```
(let ([f (λ f (x ...) body)])
  (desugar more))
```

Assume we want safety

unsafe kernel (vm)	safe kernel	vm
safe program	vm	vm
unsafe compiler	vm	safe

safe kernel

```
(f x) ⇒ if (obj_tag(f) == CLO) {
          ((clo*) f) -> code_ptr(x)
        } else { error }
```

safe program ⇒

```
if (function? f) {
  (f x)
} else { error }
```

17-3] p = ... ~~+~~ ... unsafe+

```
stdlib =
  ....
  (define (+ x y)
    (if (and (number? x)
             (number? y))
        (unsafe+ x y)
        (error)))

desugar (f x) =
  (if (fun? f) (if (= (arity f) 1)
                   (f x)
                   error)
               error)

p = .... unsafe-apply
apply f (list x y z) = (f x y z)

desugar (f x) = safe-apply f (list x)
```

174/ safety violation:

- what we wanted — ctc
- what we got — val
- who gave — pos
- who got — neg

(blame)

```
(protect ctc val pos neg)

(+
 (protect num? v p n) 7)
 => (if (num? v) v
        (error "expected num, got " v " from
                pos at neg")))
```

Desugar `(+ x y)` =>

```
(unsafe+ (protect num? x "line 27" "stdlib")
         ...)
```

17-5/
```
(define (map f l)
  (if (empty? l)
      empty
      (cons (f (first l))
            (map f (rest l)))))
```

map : (Num -> Str) x (List Num) -> (List Str)

```
protect (listof p) v pos neg =>
  checkall p v pos neg

protect (Num -> Str) f pos neg =>
  λ x.
  protect Str (f (protect x Num neg pos))
          pos neg
```

function proxy

18-1 Macro Systems

C Macros

```
#define DEBUG 1
#define MAX(x,y)
        ((x) > (y) ? (x) : (y))
#define MAX(z,x,y)
do {
    z = (x) > (y) ? (x) : (y); }
while (0);;;
```

wow Macros

Excel Macros

```
F2 => Qx721
(let  xt = (x)
      yt = (y)
  xt > yt ? xt : yt)
```

C macros are textual not, expression-oriented

```
MAX( 1?2:3 , .... )
MAX( a++ , .... )
MAX( z , xt )
```

purely substitutional

```
int db[32] = { Vi : f(i) }
```

known at compile

```
{ F(0), F(1), F(2) ...
#define F(x)  (x) * 2 + 1
```

18-2/ The language kernel should be simple and flexible...

features should be added on top of old if possible

call/cc ⇒ generator, nondet, threads, try/catch

set-box! ⇒ set! and arbitrary ds mut

λ ⇒ let

(let x = e in b)

⇒ ((λ (x) b) e)

Great languages have big desuggarers

```
(define-desugar-rules
  [(let ([x xe]) be)
   ((λ (x) be) xe)]

  [(let () be)
   be])
```

pattern

template

18-3

```
(define-desugar-rules
  [(define-desugar-rule pat tem)
   (define-desugar-rules
     [pat tem])])
```

dsrs : id × List(patr(pat, template))

let ::= "let", [< (let ([x xe]) be),
((λ (x) be) xe) >]

pattern-match : pat × se → env

transcribe : tem × env → se

pm (let ([x xe]) be) (let ([foo (+ 1 2)])
(+ foo foo))

= [x ↦ foo, xe ↦ (+ 1 2)
be ↦ (+ foo foo)]

tr ((λ (x) be) xe) =

((λ (foo) (+ foo foo)) (+ 1 2))

18-4/ pm '() '() = ∅

pm (cons pa pd) (cons ia id) =
pm pa ia ∪ pm pd id

pm var(x) in = [x ↦ in]

pm const(n) const(n) = ∅

tr '() env = '()

tr (cons ta td) env = (cons tr(ta, env)
tr(td, env))

tr var(x) env = env[x]

tr const(n) env = n

18-5/ (let* ([x * 1]
[y * 2]
[z * x + y]
[u * z + 3])
~~x~~
u)
=> (let ([x 1])
(let ([y 2])
(let ([z (+ x y)])
(let ([u (+ z 3)])
u))))

dsrs

① (let* () be) => be

② (let* ([x0 xe0] more ...) be) =>
(let ([x0 xe0]) (let* (more ...) be))

pm ② (let* ↑) =

[x0 ↦ x , xe0 ↦ 1 , be ↦ u
more ↦ MANY([y 2], [z (+ x y)], [u (+ z 3)])

tr (list tmp ...) env = map (tr tmp) (allout many env)

pm (list pat ...) in =
mergeintomany (map (pm pat) in)

[x ↦ a++ , y ↦ 7]

18-b/ desugar (or x y) =>

(let ([tmp x])
  (if *mp tmp (if x x
  y)) y))

d--

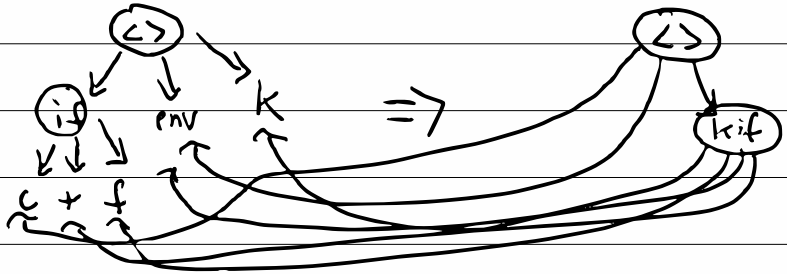
(or [redacted] 7)

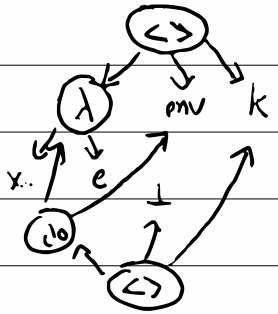
(let ([tmp y])
  (or #false tmp)) => (let ([tmp y])
                       (let ([tmp FALSE])
                         (if tmp tmp tmp)))

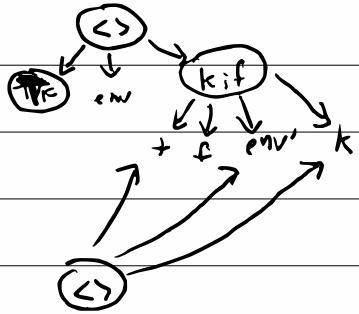
19-1 Memory Management

$\langle$ if c t f, env, k $\rangle$

$\mapsto$ $\langle$ c, env, kif(t, f, env, k) $\rangle$







19-2) $e = num \mid (op\ e\ e)$

$op = + \mid - \mid \times \mid \div$

$k = ret \mid L(op\ k\ e) \mid R(op\ num\ k)$

app(

$\langle op, e_L, e_R), k\rangle \mapsto \langle e_L, L(op, e_R, k)\rangle$

$\langle num, L(op, e_R, k)\rangle \mapsto \langle e_R, R(op, num, k)\rangle$

$\langle num_R, R(op, num_L, k)\rangle \mapsto \langle \delta(op, num_L, num_R), k\rangle$

19-3/ what should a MM do?

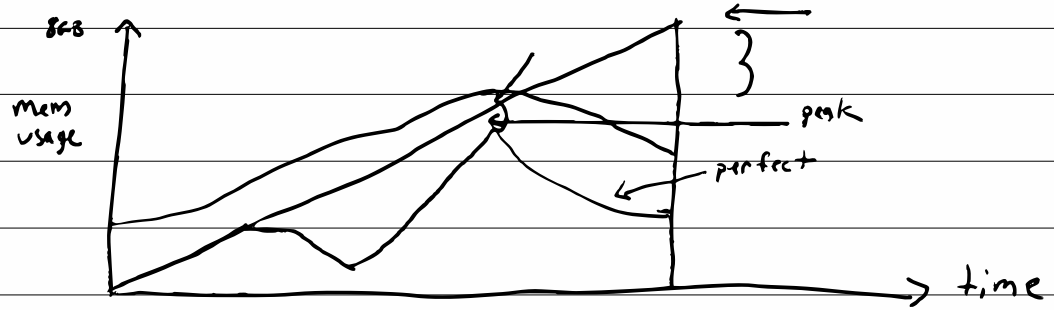
- know when to call free()

— wait to free to end (never free in between)

— free-ing "active" objects

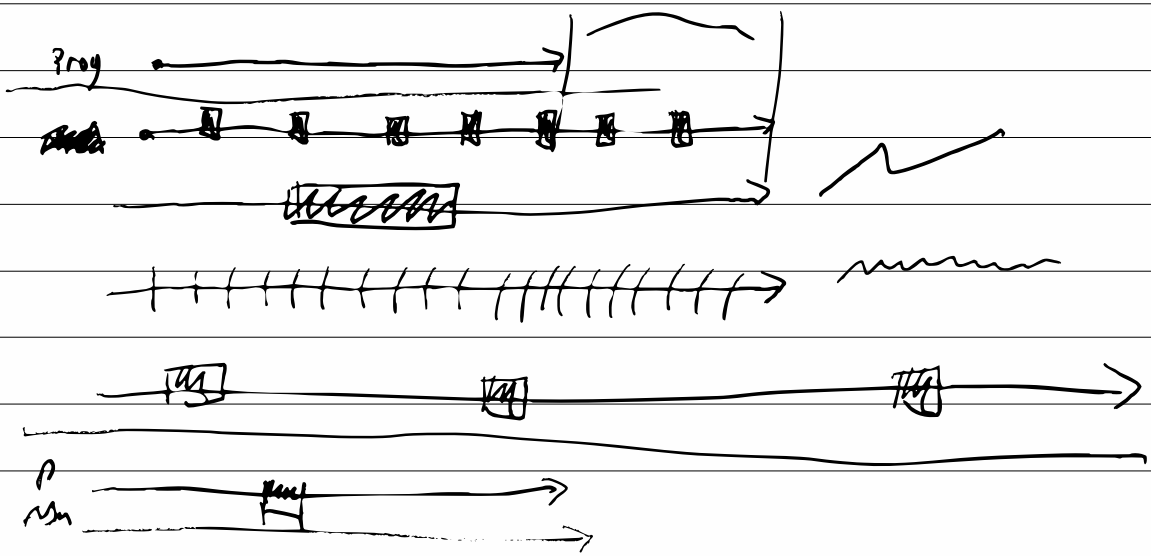
$f(x) = a$ but $= b$

Soundness = MM preserves same answer



peak overhead (min, median, avg, max)

integral

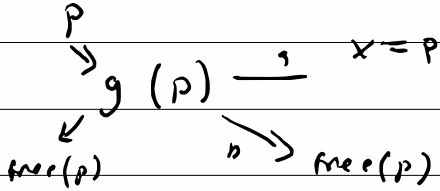


19-4/ How do you get sound MM in C?

unsound comes from aliases

one pointer — two vars/fields

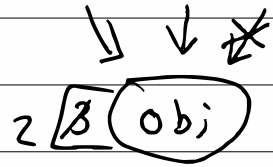
```
f() {
  char *c = ...;
  free(c);
  return; }
```



- always pass ownership
- always make copies

(Aliases)

19-5 Reference Counting / Smart Pointers

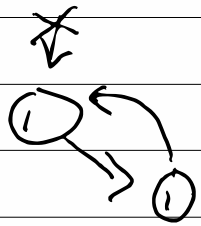



```
mkref(p) :=
    p.count++

rmref(p) :=
    if(--p.count == 0)
        free(p);
```

64-bit counts

8-bit

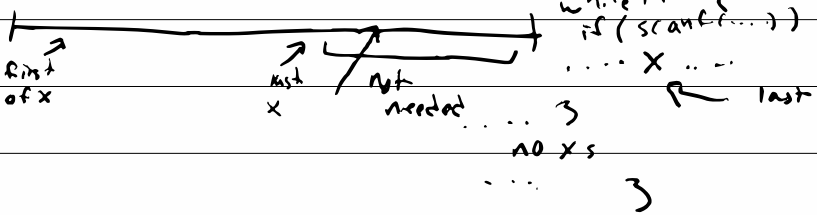


fails on cycles (leaks)

20-1/ When is an obj free()-able?

scope of var X

last use of ~~obj~~



{ int * x = malloc(...)

g = x

ret }

x is unbound

{ int x = m()

... x

free()?

if (f(7)) {

use x }

o.w. never use

}

Truth (Does such HALT??) I don't know x

vs

Provability (Do I know?) I know I don't know x

20-2 Reachability

Suppose o is an obj in memory

reach(o) iff var(o)

∨ ptr(p, o) ∧ reach(p)

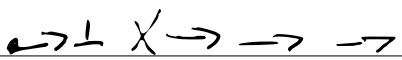
∨ field(o', o) ∧ reach(o')

∨ reg(o)

∨ stack(o)

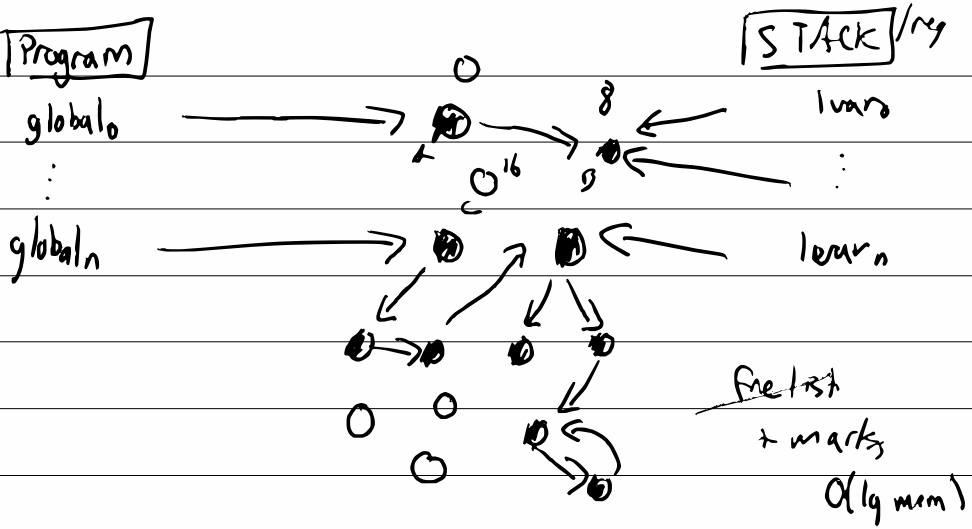
((o.f)[3].m.x)

o = NULL



unreachable objects may be freed

20-3/



A B C D E F

O(1)	malloc	~~O(lg n)~~	O(n lg n)
	free	—	
O(live)	gc	O(live) + O(mem)	
		mark, sweep	

Constraints:
- know size of objs
- all known pointers from malloc
- know obj layout

tricks:
- mark in tag
- B;B.P

John McCarthy 1969
LISP