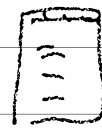
1-1

Goal: Implement a VM

What does a program mean?

"1+1" means "2"

"wordic" means 

a semantics
- english (human)
- math (universal)
- code

$\forall x,y \in \mathbb{N}.\ x + y = y + x$

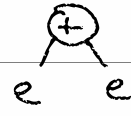
BNF or CFG

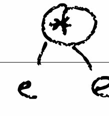
J_0: e := v | (+ e e) | (* e e)

v := number

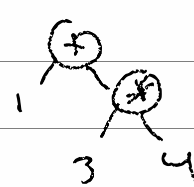
"(+ 1 (* 3 4))" $\in J_0 . e$

"(+ 1))"

e := v or 

or 

v := number



1-2/

= new Add(new Num(1),
new Mult(
new Num(3),
new Num(4)))

```
class E {}
class Add : E {}
class Num : E {}
class Mult : E {}

Add::Add(E *l, E *r) {
    this.l = l; this.r = r; }
```

(+ 1 (* 3 4))

---

semantics : meaning of programs

interpreter : a program in ~~lang~~ language M that tells you the semantics of language O

virtual machine : a fast interpreter we like

interp (a big step semantics) : $e \rightarrow v$

interp v = v

interp $(+\ e_1\ e_2)$ = $(\text{interp } e_1) +_v (\text{interp } e_2)$

$+_v$ (Num n_1) (Num n_2) = Num $(n_1 + n_2)$

Num* $\approx$ V*

1-3/ E*

```
virtual E::interp() = 0;
// interp v = v
E* Num::interp() { return this; }
// interp (+ e1 e2) = (interp e1) +v (interp e2)
E* Add::interp() {
Num* n1 = this.l.interp();
Num* n2 = this.r.interp();
  return new Num( n1.n + n2.n ); }
```

$(-\ e_1) \equiv (*\ -1\ e_1)$

$(-\ e_1\ e_2) = (+\ e_1\ (-\ e_2))$

desugar (expander) [compiler] : string tree

string expr

S-expr

Sexpr $\Longrightarrow J_0$

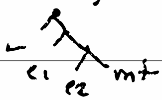
Sexpr = string or pair of Sexpr
or empty

= (- 1 2)

1 - 2

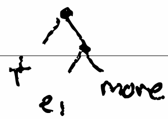
2-1/ Sexpr $\quad$ J_0

desugar $(-\ e_1) = (*\ -1\ (\text{desugar}\ e_1))$



$(-\ e_1\ e_2) = (+\ (d\ e_1)\ (d\ (-\ e_2)))$

$(+) = 0$



$(+\ e_1\ .\ \text{more}) = (+\ (d\ e_1)\ (d\ (+\ .\ \text{more})))$

$(*) = 1$

$(*\ e_1\ .\ \text{more}) = (*\ (d\ e_1)\ (d\ (*\ .\ \text{more})))$

J_1 $\quad$ e := v | (if e e e) | (e e ...)

v := number | bool | prim

prim := + | * | / | - | ≤ | < | = | > | ≥

interp v = v

interp (if e_c e_t e_f) =

c = interp e_c

ek = if c et o.w ef

return interp ek

interp (e_p e_0 ... e_n) =

p = interp e_p (must be a prim)

v_0 ... v_n = interp v_0 ... interp v_n

ret $\delta(p, v_0 \dots v_n)$

2-2

```
v* delta (prim p, list v* vs) =
  if (p == ADD)
      return new Num(vs[0].n + vs[1].n)
  if (p == LT)
      return new Bool(vs[0].n < vs[1].n)
```

---

big-step has a big problem : $e \Rightarrow v$

- it is partial
- it says nothing about "in between"
- very un-math-like and proving
- inefficient / unhelpful for implementation

small step : $e \to e$

step (if true et ef) = et

step (if false et ef) = ef

step (P v0 ... vn) = δ(P, v0 ... vn)

if step ec = ec' then step (if ec et ef)
= (if ec' et ef)

if step ei = ei' then step (e0 ... ei ei+1 ... en)
= (e0 ... ei' ei+1 ... en)

2-3)

$$(1+1) + (2+3) \overset{\text{step}}{=} 2 + (2+3) \overset{\text{step}}{=} 2 + 5 \overset{\text{step}}{=} 7$$

$$= (1+1) + 5$$

A ~~contextt~~ context, C ::= hole
| if C e e
| if e C e
| if e e C
| $(e \ldots C\ e \ldots)$

plug : $C \times e \Rightarrow e$

plug hole e = e

plug (if C e_1 e_2) e_p = (if ~~e_p~~ (plug C e_p) e_1 e_2)

plug $(e_b \ldots C\ e_a \ldots)$ e_p = $(e_b \ldots (\text{plug } C\ e_p)\ e_a \ldots)$

plug $(\blacksquare + (2+3))$ $(1+1)$ = $(1+1) + (2+3)$

C $\qquad$ 2 = $2 + (2+3)$

parse : $e \Rightarrow C \times e$

2-4

$$e \Rightarrow e$$

step $C[\text{if true } et\ ef] = C[et]$

setp $C[\text{if false } et\ ef] = C[ef]$

step $C[p\ v_0 \ldots v_n] = C[\delta(p, v_0 \ldots v_n)]$

find_redex : $e \Rightarrow C \times e$

$\uparrow$ redex

reducible expression

When do two programs mean the same thing?

"1 + 2" "2+1"

"4 billion + 1 + 2" "2 + 1 + 4 billion"

$\underline{2}$ $C = ▨$

"quicksort" "mergesort" "heapsort"

"insertion sort"

$\forall l.$ "mergesort l" = "hs l" — $C = (▨\ L)$

$\forall C.$ $\overset{\text{interp}}{C[x]} = \overset{\text{interp}}{C[y]}$ — observational equivalence

time e = e's ans $\times$ secs

3-1 step : $e \Rightarrow e$ $\quad \rightarrow \rightarrow \rightarrow \quad v$

$C[\text{if true } e_t \ e_f] \rightarrow C[e_f]$

.... + ... * ... − .... (1+1) .... * .... +

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

evaluation context, E

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

$E[\text{if true } e_t \ e_f] \rightarrow E[e_t]$

$E[\text{if false } e_t \ e_f] \Rightarrow E[e_f]$

$E[(p\ v_0 \ldots v_n)] \rightarrow E[\delta(p, v_0 \ldots v_n)]$

find-redex : $e \Rightarrow (E, e)$ or #false

find-redex v = false

fr (if $e_0\ e_1\ e_2$) = case (fr e_0) with

#false $\Rightarrow$ (hole, (if $e_0\ e_1\ e_2$))

$(E, e_0') \Rightarrow$ (if $E\ e_1\ e_2$, e_0')

3-21

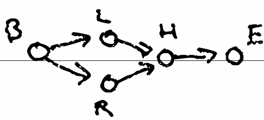
fr (v ... e_0 e_1 ... e_n) =

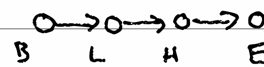
  (E, e_0') = fr e_0

  ((v ... E e_1 ... e_n), e_0')

  new EApp(v, E, e_1 .. e_n)

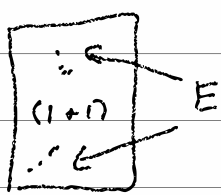
fr (v ...) = (hole, (v ...))

step$_C$ 

step$_E$ 

The Standard Reduction theorem

Alonzo Church   ((Church-Rosser))   Rosser



E = (if (* 7 8 (+ 1 2 (* ....

  ..... (if .... hole ...)$^{(A34)}$ ...))

$\underbrace{E[(1+1)]}_{\text{2n time, 1n space}} \rightarrow \underbrace{E\ [2]}_{\text{2n time, 1n space}}$

3-3) $\quad step_E : e \Rightarrow e$

machine semantics

$$
\begin{tikzcd}
e \arrow[r, "step_e"] \arrow[d, "inject"'] & e' \arrow[r, "s"] & e'' \arrow[r] & e''' \arrow[r] & v \\
st \arrow[r, "step_{st}"'] & st' \arrow[u, "extract"'] \arrow[r] & \arrow[r] & st''' \arrow[u, "extract"'] \arrow[r] & st'''' \arrow[u, "extract"']
\end{tikzcd}
$$

CC0 $\quad st = \langle e, E \rangle$

$\quad$ inject $e = \langle e, \text{hole} \rangle$

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

$\quad$ done? $\langle v, \text{hole} \rangle = \text{true}$

1 $\langle \text{if } e_c\ e_t\ e_f, E \rangle \Rightarrow \langle e_c, E[\text{if hole } e_t\ e_f] \rangle$

2 $\langle \text{true}, E[\text{if hole } e_t\ e_f] \rangle \Rightarrow \langle e_t, E \rangle$

3 $\langle \text{false}, E[\text{if hole } e_t\ e_f] \rangle \Rightarrow \langle e_f, E \rangle$

4 $\langle e_0\ e_1 \ldots e_n, E \rangle \Rightarrow \langle e_0, E[(\text{hole } e_1 \ldots e_n)] \rangle$

5 $\langle v, E[v_0 \ldots \text{hole } e_0\ e_1 \ldots] \rangle \Rightarrow \langle e_0, E[(v_0 \ldots v \text{ hole } e_1 \ldots)] \rangle$

6 $\langle v_n, E[p\,v \ldots \text{hole}] \rangle \Rightarrow \langle \delta(p, v_0 \ldots v_n), E \rangle$

$(4n, 2n) \longrightarrow (1n, 1n)$

3-4/

HL: (test '(+ 1 (* 2 (if true 3 4)))
7)

test (new Sexpr (new Atom("+"),
new Se(new A ("1"),
.....
),
new Num (7))

test (se, ex-val) ==
e = desugar se
big-step eval (e) = actual-big-val
if (abv != ev) { error }
small-step eval (e) = ac - sm - v
(if (asv != ev) { error }
cco-eval e = accv
if (accv != ev) { error }
ll-eval e = allv
if (allv != ev) { error }

3-5

low level-eval e =

print e as C constructors into "x.c"

compile "x.c" into "x.bin"

run "x.bin" and capture output

pause output

return value

cc x.c ll.c -o x.bin

9-11 $CC_0 \rightarrow CK_0$

$st = \langle e, k \rangle$

$k = kret$ // hole

continuation kontinuation stac k

$kif\ e\ e\ k$ // if E e e

$kapp\ (v\ldots)\ (e\ldots)\ k$ // (v… E e…)

$A \Rightarrow kapp\ (v\ldots)\ (e\ldots)\ B$

$\Leftrightarrow B[(v\ldots\ A\ e\ldots)]$

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

$extract\ \langle e, k \rangle = kintoE(k)[e]$

$done?\ \langle v, kret \rangle = true$

$\langle if\ e_c\ e_t\ e_f, k \rangle \mapsto \langle e_c, kif(e_t, e_f, k) \rangle$

$\langle true, kif(e_t, e_f, k) \rangle \mapsto \langle e_t, k \rangle$

$\langle false, kif(e_t, e_f, k) \rangle \mapsto \langle e_f, k \rangle$

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

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

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

$\delta(SUB, (v_1\ v_0)) = v_0 - v_1$

$(3\ 4\ 5)$

5-1/ $J_1 \Rightarrow J_2$

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

$x ::=$ variable names

$v ::= b \mid f$

new

$b ::=$ number | bool | prim

$f ::=$ function names

$p ::= (\text{program } (d \ldots)\ e)$

$d ::= (\text{define } (f\ x \ldots)\ e)$

(program (define (add1 x) (+ 1 x))
(add1 5))

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

J_1: step : $e \Rightarrow e$

J_2: step : $\vec{\Delta} \times e \Rightarrow e$

$(f \mapsto d)$

$$E[(f\ v_0 \ldots)] \mapsto E[e[x_0 \leftarrow v_0] \ldots [x_n \leftarrow v_n]]$$

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

5-2/ $e[x \leftarrow v]$ means look inside of e, find all the xs and replace with v

$x[x \leftarrow v] = v$

$y[x \leftarrow v] = y \quad (y \in \overset{\text{vars}}{X})$

$u[x \leftarrow v] = u \quad (u \in \text{values})$

$(\text{if } e_1 \; e_2 \; e_3)[x \leftarrow v] = (\text{if } e_1[x \leftarrow v] \; e_2[x \leftarrow v] \; e_3[x \leftarrow v])$

$(e_0 \ldots e_n)[x \leftarrow v] = (e_0[x \leftarrow v] \ldots e_n[x \leftarrow v])$

$f(x) = 7x + 2x^2 + 1$

$f(5) = 7 \cdot 5 + 2 \cdot 5^2 + 1$

```
(define (f x y) (+ (* x 2) (* -1 y)))
(define (g x) (f x x))                 ] = Δ
(+ 5 (g 10) (f 9 (g 1)))                 = e
(  "  (f 10 10)        "  )
(  "  (+ 10·2 -10)     "  )
(+ 5   10              "  )
(+ 5   10    (f 9 1))
(            (+ 9·2 -1))
(+ 5   10    17)
   32
```

5-3/ $CK_0 \Rightarrow CK_1$ st = $\langle \Delta, e, k \rangle$

$(1+N)$

$$\langle \Delta, v_n, kapp((f\ v_0 ..), ()), k \rangle^N$$
$$\mapsto \langle \Delta, e[x_0 \leftarrow v_0] \dots [x_n \leftarrow v_n], k \rangle$$
where $\Delta(f) = (\text{define } (f\ x_0 \dots x_n)\ e)$

$$\langle \Delta, (\text{if } e_c\ e_t\ e_f), k \rangle \mapsto \langle \Delta, e_c, kif(e_t, e_f, k) \rangle$$

(if (x > 10 mi) (error)
(+)
(define (f x) (f x)))
(f 0)

$\mapsto (f\ \overset{1}{0}) \mapsto (f\ \overset{2}{0})$

"Jay"

proper function call implementation

"Guido"

tail-call optimization

6-1 CK_1 is linear-time, so it's not fast :(
and unrealistic because syntax is available at run-time

goal: machine with constant function calls

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

"x" $\quad f(x) = 2x^2 + 5x + 7$

$\langle f(5) \rangle \mapsto \langle 2x^2 + 5x + 7, [x \leftarrow 5] \rangle$

$\langle 2 \cdot 5^2 + 5x + 7, [x \leftarrow 5] \rangle$

$\langle 50 + 5 \cdot 5 + 7, [x \leftarrow 5] \rangle$

$\langle 82, [x \leftarrow 5] \rangle \quad = 82$

6-2) $CK_1 \Rightarrow CEK_0$

$st ::= \langle \Delta, e, env, k \rangle$

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

Jay's env as c = make-v-empty | make-v-cons (make-v-cons (x,v), env)

$inject^{\Delta} e = \langle \Delta, e, mt, kret \rangle$

$extract \langle \Delta, e, \cancel{e}env, k \rangle = E(k)[e[env\ replace]]$

$done? \langle \Delta, v, env, kret \rangle$

!!! ↓ WRONG ↓ !!! (look ahead 2 pgs)

$\langle \Delta, \cancel{x}, mt, k \rangle \mapsto error$

$\langle \Delta, x, env[x \leftarrow v], k \rangle \mapsto \langle \Delta, v, \overset{env[x \leftarrow v] \text{ or}}{mt}, k \rangle$

$\langle \Delta, x, env[y \leftarrow v], k \rangle \mapsto \langle \Delta, x, env, k \rangle$

$\langle \Delta, (if\ e_c\ e_t\ e_f), env, k \rangle \mapsto$

$\langle \Delta, e_c, env, kif(e_t, e_f, k) \rangle$

$\langle \Delta, true, env, kif(e_t, e_f, k) \rangle \mapsto \langle \Delta, e_t, env, k \rangle$

$\langle \Delta, false, env, kif(e_t, e_f, k) \rangle \mapsto \langle \Delta, e_f, env, k \rangle$

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

$\langle \Delta, v_n, env, kapp((v_0 \ldots), (e_0\ e_1 \ldots), k) \rangle$

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

$\langle \Delta, v_n, env, kapp((p\ v_0 \ldots), (), k) \rangle \mapsto \langle \Delta, \delta(p, (v_0 \ldots v_n)), env, k \rangle$

$\langle \Delta, v_n, env, kapp((f\ v_0 \ldots), (), k) \rangle \mapsto$ let $(define\ (f\ x_0 \ldots)\ e) = \Delta(f)$

$\langle \Delta, e, \cancel{env}[x_0 \mapsto v_0] \ldots, k \rangle$

6-3) (define (f x) (+ x y))
(define (g y) (f 5))
(g 10)

(g 10) $\rightarrow$ (f 5) $\rightarrow$ (+ 5 y)

$\langle (g\ 10), \emptyset \rangle \mapsto \langle (f\ 5), mt[y \leftarrow 10] \rangle$

$\mapsto \langle (+\ x\ y), mt[y \leftarrow 10][x \leftarrow 5] \rangle$

$\mapsto \langle (+\ 5\ y),$ " $\rangle$

$\mapsto \langle (+\ 5\ 10),$ " $\rangle$

$\mapsto \langle 15,$ " $\rangle$

WRONG:

$\langle \Delta, (\text{if } e_c\ e_t\ e_f), env^1; k \rangle$

$\mapsto \langle \Delta\ e_c, env^1; kif(e_t, e_f, k) \rangle$

$\langle \cancel{true}\ \Delta, true, env^2; kif(e_t, e_f, k) \rangle$

$\mapsto \langle \Delta, e_t, env^2; k \rangle$

(define (g y) true) Dynamic Scope

(define (f y) (if (g 10) y 6))

(f 6)

6-4/ $k ::= \text{kret} \mid \text{kif}(e, e, env, k)$

$\mid \text{kapp}((v \ldots), (e \ldots), env, k)$

RIGHT

$\langle \Delta, x, env, k \rangle \mapsto \langle \Delta, env(x), mt, k \rangle$

$\langle \Delta, \text{if } e_c\, e_t\, e_f, env, k \rangle \mapsto \langle \Delta, e_c, env, \text{kif}(e_t, e_f, env, k) \rangle$

$\langle \Delta, \text{true}, \overset{\text{NEW}}{env'}, \text{kif}(e_t, e_f, \overset{\text{OLD}}{env}, k) \rangle \mapsto \langle \Delta, e_t, \overset{\text{OLD}}{env}, k \rangle$

$\langle \Delta, \text{false}, \_\_, \text{kif}(e_t, e_f, env, k) \rangle \mapsto \langle \Delta, e_f, env, k \rangle$

$\langle \Delta, (e_0\ e_1 \ldots), env, k \rangle \mapsto \langle \Delta, e_0, env, \text{kapp}((), (e_1 \ldots), env, k) \rangle$

$\langle \Delta, v_n, \_\_, \text{kapp}((v_0 \ldots), (e_0\ e_1 \ldots), env, k) \rangle$

$\mapsto \langle \Delta, e_0, env, \text{kapp}((v_0 \ldots v_n), (e_1 \ldots), env, k) \rangle$

$\langle \Delta, v_n, \_\_, \text{kapp}((p\ v_0 \ldots), (), \_\_, k) \rangle$

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

$\langle \Delta, v_n, \_\_, \text{kapp}((f\ v_0 \ldots), (), \_\_, k) \rangle$

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

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

6-5/ $J_2 \rightarrow J_3$

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

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

$x ::= \text{variable names}$

$b ::= \text{number} \mid \text{bool} \mid \text{prim}$

$E[((\lambda\ (x_0 \ldots x_n)\ e)\ v_0 \ldots v_n)]$

$\mapsto E[\, e[x_0 \leftarrow v_0] \ldots [x_n \leftarrow v_n]\,]$

capture-avoiding

$((\lambda\ (x)\ x)\ 5) \mapsto 5$

$(((\lambda\ (x)\ (\lambda\ (y)\ x))\ 5)\ 6) \mapsto$

$((\lambda\ (y)\ 5)\ 6) \mapsto 5$

$(((\lambda\ (f)\ (\lambda\ (y)\ \boxed{(f\ 5)}))\ (\lambda\ (x)\ y))\ 6)\ 7) \mapsto$

$(((\lambda\ (y)\ (\lambda\ (x)\ y))\ 6)\ 7) \mapsto$

$((\lambda\ (x)\ 6)\ 7) \mapsto 6$

6-6

$\langle v_n, \overset{\text{new}}{env}, kapp(((\lambda\ (x_0 \dots x_n)\ e)\ v_0 \dots), (), env, k)\rangle$

$\mapsto \langle e, mt[x_0 \leftarrow v_0] \dots [x_n \leftarrow v_n], k\rangle$

$\langle ((\lambda x. \lambda y. x)\ 5)\ 6,\ \emptyset,\ kret\rangle$

$\langle \lambda x. \lambda y. x,\ \emptyset,\ kapp((), (5\ \cancel{6}), \emptyset, \cancel{kret})\rangle$ kapp((), (6), $\emptyset$, kret))

$\langle 5, \emptyset, kapp((\lambda x. \lambda y. x), (), \emptyset, kapp((), (6), \emptyset, kret))\rangle$

$\langle \lambda y. x, \emptyset[x \leftarrow 5], kapp((), (6), \emptyset, kret)\rangle$

$\langle 6, \emptyset, kapp((\lambda y. x), (), \emptyset, kret)\rangle$

$\langle x, \emptyset[y \leftarrow 6], kret\rangle$ $\quad$ $\langle x, \emptyset[x \leftarrow 5][y \leftarrow 6], kret\rangle$

old ↑ $\quad$ $\langle 5, mt, kret\rangle$

$\rightarrow 5$

$((\lambda x. \lambda y. x)\ 5)\ 6) \mapsto (\lambda y. 5)\ 6 \mapsto 5$

CEK

old $v ::= num \mid bool \mid prim \mid (\lambda\ (x \dots)\ e)$

new $v ::= num \mid bool \mid prim \mid closure((\lambda (x \dots) e), env)$

$\langle (\lambda (x \dots) e), env, k\rangle \mapsto \langle closure((\lambda (x \dots) e), env), \emptyset, k\rangle$

$\langle v_n, -, kapp((closure((\lambda\ (x_0 \dots x_n)\ e), env), v_0 \dots), (), \cancel{k})\rangle$

$\mapsto \langle e, env[x_0 \leftarrow v_0] \dots [x_n \leftarrow v_n], k\rangle$

6-7/

desugar (let ([x0 e0] .... [xn en]) be)
= ((λ (x0 ... xn) be) e0 ... en)

(let ([x 5] [y 6])
(+ x y))
=
((λ (x y) (+ x y))
5 6)

(let ([x ← 5])
(+ x
(let ([x (+ 1 x)]) (+ x x))
(+ x 4)))
∅[x←5][x←6]
= (+ 5
(let ([x ← (+ 1 5)]) (+ ~~x~~ x))
(+ 5 4))
= (+ 5 (+ 6 6) (+ 5 4))
= (+ 5 12 9)
= 26

6-8/

```
desugar (let* () be) = be
desugar (let* ([x0 e0] [x1 e1] ...) be)
  = (let ([x0 e0])
      (let* ([x1 e1] ...) be)
```

```
(let* ([x 5]
       [y (+ x 1)]
       [z (+ x y)])
  (+ z z))
```

```
normal let: ((λ (x y z) (+ z z))
             5 (+x 1) (+x y))

let*: ((λ(x) ((λ(y) ((λ(z) (+ z z)) (+ x y)))
        (+ x 1))) 5)
```

7-1/ ((let ([x ← 5])
(λ (y) (+ x y)))
6) ⇒ (+ 5 6)

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

id : $\lambda x.x$

$(\lambda x.\lambda y.x)\ (\lambda z.z)\ (\lambda g.g)$

$\Rightarrow$

$(\lambda z.z)$

Alonzo Church's Lambda Calculus

Church encoding == Object-oriented Prog.

represent as interface   interface vs representation

bool : Opt 1 → Opt 2 → Some Option 1 or 2

true := $\lambda x.\lambda y.\ x$

false := $\lambda x.\lambda y.\ y$

if := $\lambda b.\lambda t.\lambda f.\ b\ t\ f$

if true M N ⇒ $(\lambda btf.\ b\ t\ f)$ true M N

⇒ true M N ⇒ $(\lambda xy.x)$ M N ⇒ M

7-2/ Church-encoded numbers

numbers are iteration

num : Thing To Do ⇒ Something to do it to
⇒ Result of doing it N times

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

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

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

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

add1 zero $\Rightarrow \lambda f. \lambda z.\ f\ (\text{zero}\ f\ z)$
$\Rightarrow \lambda f. \lambda z.\ f\ z$ = one

plus := $\lambda n. \lambda m. \lambda f. \lambda z.\ m\ f\ (n\ f\ z)$

plus one one $\Rightarrow \lambda f. \lambda z.\ \text{one}\ f\ (\text{one}\ f\ z)$
$\Rightarrow \lambda f. \lambda z.\ \text{one}\ f\ (f\ z)$
$\Rightarrow \lambda f. \lambda z.\ f\ (f\ z)$ = two

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

mult two two $\Rightarrow \lambda f. \lambda z.\ \text{two}\ (\text{two}\ f)\ z$
$\lambda f. \lambda z.\ (\text{two}\ f)\ (\text{two}\ f)\ z$
$\lambda f. \lambda z.\ f\ (f\ (f\ (f\ z)))$ =
four

7-3/ zero? $= \lambda n.\ n\ (\lambda x.\ false)\ true$

$zero?\ zero \Rightarrow zero\ (\lambda x.\ F)\ T$

$\Rightarrow T$

$zero?\ one \Rightarrow one\ (\lambda x.\ F)\ T$

$\Rightarrow (\lambda x. F)\ T$

$\Rightarrow F$

$fst\ (pair\ M\ N) \Rightarrow M$

$snd\ (pair\ M\ N) \Rightarrow N$

$pair := \lambda x. \lambda y. \lambda sel.\ sel\ x\ y$

$fst := \lambda p.\ p\ true$

$snd := \lambda p.\ p\ false$

$fst\ (pair\ M\ N) \Rightarrow (pair\ M\ N)\ true$

$\Rightarrow (true\ M\ N) \Rightarrow M$

$select\ (inl\ M)\ f\ g \Rightarrow f\ M$

$select\ (inr\ N)\ f\ g \Rightarrow g\ N$

$inl := \lambda v.\ \lambda f. \lambda g.\ f\ v$

$inr := \lambda v.\ \lambda f. \lambda g.\ g\ v$

$select := \lambda o. \lambda f. \lambda g.\ o\ f\ g$

copy := λn. n add1 zero

7-4/ sub1 := λn. fst (n F (pair zero zero))

F := λp. pair (snd p) (add1 (snd p))

sub := λn. λm. m sub1 n

fac := λn.

defn

if (zero? n)

copy, not a reference

one

(mult n (fac (sub1 n)))

mkfac := λfac.

(λn. if (zero? n) one

ref

(mult n (fac (sub1 n)))))

fac := Z mkfac

Z mkfac ⇒ fac

Z (λfac. M) ⇒ M [fac ← (Z mkfac)]

Z (λx. M) ⇒ M (Z (λx. M))

← fixed-point combinator (eager) aka Y

Z := λf. ((λx. (f (λv. (x x v))))

(λx. (f (λv. (x x v)))))

Z f = f (Z f)

Z-5/ $\Omega := \omega \; \omega$

$\omega := \lambda x.\; x \; x$

$\Omega \Rightarrow \omega \; \omega \Rightarrow$

$(\lambda x.\; (x \; x)) \; (\lambda x.\; (x \; x))$

$\Rightarrow (\lambda x.\; (x \; x)) \; (\lambda x.\; (x \; x))$

$\Rightarrow \omega \; \omega$

$\Rightarrow \Omega$

The Lambda Calculus $\subseteq$ J_2

A J_2 to convert Church to Normal :=

Church2Normal := $\lambda n.\; n \; (\lambda x.\; (+ \; 1 \; x)) \; 0$

Church2Normal (fac (succ (plus two two)))

$\Rightarrow 120$

8-1/ J_2:
```
(define (even? x)
  (if (zero? x) true
      (odd? (sub1 x))))
(define (odd? x)
  (if (zero? x) false
      (even? (sub1 x))))
(even? 10)
```
$\Delta = \{$ even? , odd? $\}$

odd? is undefined

J_3:
```
(let* ([even? (λ (x) .... odd? ...)]
       [odd? (λ (x) .... even? ...)])
  (even? 10))
```

$J_3 \Rightarrow J_4$ = v ::= .... | (λ x (x ...) e)

recursive name of fun

```
(let ([fac (λ ifac (n) (if (= n 0) 1
                           (* n (ifac (- n 1)))))])
  (fac 5))
```

(desugar (λ (x ...) e)) ⇒ (desugar (λ rec (x ...) e))

8-2/ OLD:

$$E[\ell \; v_0 \cdots v_n] = E[e[x_0 \leftarrow v_0] \cdots [x_n \leftarrow v_n]]$$

where $\ell = (\lambda\ (x_0 \cdots x_n)\ e)$

NEW:

$$E[\ell \; v_0 \cdots v_n] = E[e[f \leftarrow \ell][x_0 \leftarrow v_0] \cdots [x_n \leftarrow v_n]]$$

where $\ell = (\lambda\ f\ (x_0 \cdots x_n)\ e)$

$CEK_1 \Rightarrow CEK_2$

OLD: $\langle (\lambda\ (x_0 \cdots x_n)\ e), env, k \rangle$

$\mapsto \langle clo((\lambda\ (x_0 \cdots x_n)\ e), env), mt, k \rangle$

$\langle \ell, env, k \rangle \mapsto \langle clo(\ell, env), mt, k \rangle$

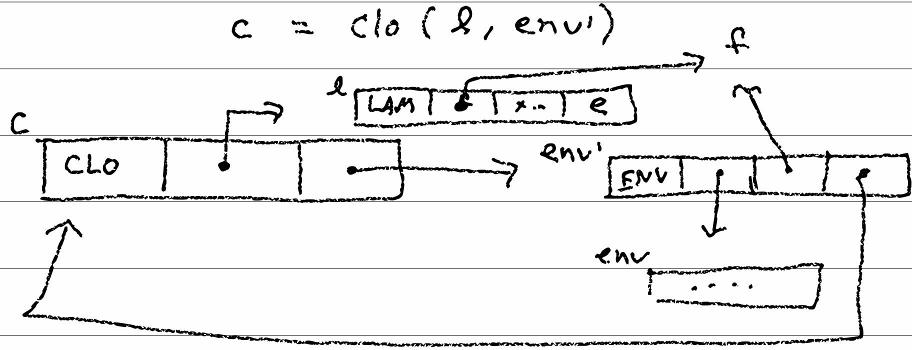
where $\ell = (\lambda\ (x_0 \cdots x_n)\ e)$

NEW: $\langle \ell, env, k \rangle \mapsto \langle \quad c \quad , mt, k \rangle$

where $\ell = (\lambda\ f\ (x_0 \cdots x_n)\ e)$

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

$c = clo(\ell, env')$



8-3/ envp = make_env_ext (env, l→f, NULL);

clo = make_clo (l, envp);

envp → val = clo; // ← installs cycle!

nat-unfold :=

(λ rec (f z n)

(if (= n 0) z

(f n (~~nat-unfold~~ rec f z (- n 1)))))

fac n = nat-unfold (λ (n a) (* n a)) 1 n

9-1/ data structures

$v = \ldots \mid p$ (primitives) $\mid b$ (constants)

Numbers, $7 \in b$, $+ \in p$

Unit: $tt \in b$ // Void void in C

Pairs: pair, fst, snd $\in p$ (pairv v_1 v_2) $\in V$ $\forall v_1, v_2$

"and"

(pair 1 2)

$E[(\text{pair } v_1\ v_2)] \to E[(\text{pairv } v_1\ v_2)]$

$E[(\text{fst } (\text{pairv } v_1\ v_2))] \to E[v_1]$

$E[(\text{snd } (\text{pairv } v_1\ v_2))] \to E[v_2]$

Variants: "or" List = empty OR node

Unit or (data x list)

$v = \ldots \mid (\text{inl } v) \mid (\text{inr } v)$

$p = \ldots \mid \text{inl} \mid \text{inr}$

$e = \ldots \mid \text{case } e \text{ as } [(\text{inl } x) \Rightarrow e]\ [(\text{inr } x) \Rightarrow e]$

$E = \ldots \mid \text{case } E \text{ as } [(\text{inl } x) \Rightarrow e]\ [(\text{inr } x) \Rightarrow e]$

$E[\text{case } (\text{inl } v) \text{ as } [(\text{inl } x_1) \Rightarrow e_1][(\text{inr } x_2) \Rightarrow e_2])$

$\mapsto E[e_1[x_1 \leftarrow v]]$

9-2/ CEK ...

$K = \ldots \mid \text{casek}\ env\ x_1\ e_1\ x_2\ e_2\ k$

$\langle \text{case } e_s \text{ as } [(\text{inl } x_1) \Rightarrow e_1]\ [(\text{inr } x_2) \Rightarrow e_2], env, k \rangle$

$\mapsto \langle e_s, env, \text{casek}\ env\ x_1\ e_1\ x_2\ e_2\ k \rangle$

$\langle \text{inl } v, \_, \text{casek}\ env\ x_1\ e_1\ x_2\ e_2\ k \rangle$

$\mapsto \langle e_1, env[x_1 \leftarrow v], k \rangle$

$\langle \text{inr } v, \_, \text{casek}\ env\ x_1\ e_1\ x_2\ e_2\ k \rangle$

$\mapsto \langle e_2, env[x_2 \leftarrow v], k \rangle$

Bool = Unit or Unit

true = inl tt

false = inr tt

if c t f = case c as [(inl _) ⇒ t]
[(inr _) ⇒ f]

X or Y = Pair Boolean (X ∪ Y)

inl v = pair #f v

inr v = pair #t v

case es as [(inl x_1) ⇒ e_1] [(inr x_2) ⇒ e_2] =

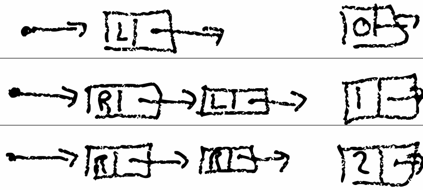
(let ([vs es]) (if (fst vs) (let ([x_1 (snd vs)]) e_1)
(let ([x_2 (snd vs)]) e_2)))

9-3/ Shape = Circle or (Rect or Triangle)

○ = inl ....

□ = inr (inl ...)

Δ = inr (inr ...)



Algebraic Data Types

Type = 1 // unit ++

0 // Nothing

Type × Type // Pair (pair e_1 e_2)

Type + Type // Variant (inl e_1)(inr e_2)

B // Base types 5

B = Int32 | ...

Bool = 1 + 1 Nat = 1 + Nat 110

true = inl ++ 0 zero = inl ++ two = inr (inr (inl ++))

false = inr ++ 10 one = inr (inl ++)

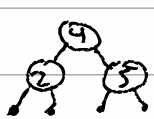
List A = 1 + (A × List A)

mt = inl ++ [3, 4] = inr (pair 3

[1] = inr (pair 1 (inl ++)) inr (pair 4

inl ++))

Tree A = 1 + (Tree A × A × Tree A)



= inr (pair (inr (pair (inl ++) (pair 2 (inl ++))))

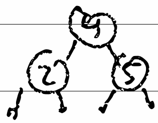
(pair 4 (inr (pair (inl ++) (pair 5 (inl ++))))))

9-4/

leaf = inl tt

node t_1 v t_2 = inr (pair t_1 (pair v t_2))

single v = node leaf v leaf

 = node (single 2) 4 (single 5)

Option type : Maybe A = None | Just A

$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad 1 + A$

nothing = inl tt

just v = inr v

hash-ref : Hash K V × K → Maybe V

Hash ID (Maybe People) × ID → Maybe (Maybe People)

seq : Maybe A × (A → Maybe B) → Maybe B

seq ma f = case ma as
[None ⇒ None]
[Just a ⇒ f a]

seq (hash-ref ht 17)
(λ (x) (hash-ref ht2 x))

9-5/ List A = 1 + Pair A (List A)

empty = inl tt
cons a d = inr (pair a d)
1, 2, 3 = cons 1 (cons 2 (cons 3 empty)))

map f l =
case l as
[(inl t) => empty]
[(inr x) => cons (f (fst x)) (map f (snd x))

map add1 (1, 2, 3) => (2, 3, 4)
filter even? (2, 3, 4) => (2, 4)

fold f z l =
case l as
[empty => z]
[cons a d => f a (fold f z d)]
sum l = fold + 0 l
sum (1, 2, 3) = (+ 1 (+ 2 (+ 3 0)))
map f l = fold (λ (a d) (cons (f a) d)) empty l

9-6/ fold fusion

$$\text{fold } f_1 \; z_1 \; (\text{fold } f_2 \; z_2 \; l)$$

$$=$$

$$\text{fold } (\lambda \; (a \; d_{12}) \; f_1 \; \text{fst } (f_2 \; a \; (\text{snd } d_{12})) \; \text{snd}) \; (z_1, z_2) \; l$$

$$\text{map } f_1 \; (\text{map } f_2 \; l) = \text{map } (f_1 \circ f_2) \; l$$

sum ↗ sum ↗

Haskell

90-1/

```
int x = 7;
int y = x;
x = x + 1;
return x-y;        => 1
```

```
(let* ([x 7]
       [y x])
  ....
  (- x y))
```

In C, a variable is a container storing a value

In JS, a variable is a named value

A box is a container that might change

p = ... | box | unbox | set-box!

```
(let* ([xb (box 7)]
       [y (unbox xb)])
  (set-box! xb (+ (unbox xb) 1))
  (- (unbox xb) y))              => 1
          =subst=>
(set-box! (box 7) (+ (unbox (box 7)) 1))
(- (unbox (box 7)) (unbox (box 7)))))        => 0
```

```
(let ([yb (let ([xb (box 0)])
            (set-box! xb (+ 7 (unbox xb)))
            xb)])
  (unbox yb))        => 7
```

10-2/

```
(define (make-counter)
    (let ([cb (box 0)])
      (λ ()
        (set-box! cb (+ 1 (unbox cb)))
        (unbox cb))))
(let ([c1 (make-counter)]
      [c2 (make-counter)])
  (list (c1) (c2) (c2) (c2) (c1)))
  => (list 1 1 2 3 2 )
```

Semantics of J_6

$v = \ldots \mid \sigma$ (pointers)

OLD $\Rightarrow : e \to e$ $\quad E[$~~[illegible]~~ $(\text{if true } e_t\ e_f)] \Rightarrow E[e_t]$

NEW $\Rightarrow : \Sigma \times e \to \Sigma \times e$

$\Sigma : \sigma \to v$

$\Sigma \times E[\text{if true } e_t\ e_f] \Longrightarrow \Sigma \times E[e_t]$

$\Sigma \times E[\text{box } v] \Rightarrow \Sigma[\sigma \mapsto v] \times E[\sigma]$ where σ is fresh

$\Sigma \times E[\text{unbox } \sigma] \Rightarrow \Sigma \times E[\Sigma[\sigma]]$

$\to$ C

$\Sigma \times E[\text{set-box! } \sigma\ v] \Rightarrow \Sigma[\sigma \mapsto v] \times E[\overset{v}{\#}]$

not C way

10-3/ option 1: $CEK_3 \Rightarrow CESK_0$

CESK $st = \langle e, env, sto, k \rangle$

$sto = mt \mid sto[\sigma \mapsto v]$

$\langle$ if ec et ef, env, sto, k $\rangle \mapsto \langle$ ec, env, sto, ifk env et ef k $\rangle$

$\langle v, -, sto,$ appk (box) () — k $\rangle$

$\mapsto \langle \sigma, mt, sto[\sigma \mapsto v], k \rangle$

option 2: box is a primitive that does

(box v) = [V_BOX, ptr to v]

(unbox (box v)) = ret ptr

(set-box! [V_BOX, ptr$_1$] ptr$_2$) = changes to memory

[V_BOX, ptr$_2$]

```
obj_t* delta_setbox ( obj_t* args ) {
((vbox_t*) second( args )) -> v = first(args);
    return make_v_void(); }
```

10-4/ Option 1:

p = ... | set-fst! | set-snd!

```
(let ([p (pair 1 2)])
  (set-fst! p 3)
  (fst p))   =>  3
```

Option 2:

```
mpair x y = pair (box x) (box y)
mfst p = unbox (fst p)
mset-fst! p nx = set-box! (fst p) nx
```

```
desugar (begin) = (void)   // ++
desugar (begin e) = e
desugar (begin e x ...) = (let ([_ e]) (begin x ...))
desugar (begin0 e x ...) = (let ([y e])
                              (begin x ... y))
desugar (when c e ...) = (if c (begin e ...) void)
desugar (unless c e ...) = (when (not c) e ...)
desugar (while c e ...) =
  ((λ loop () (when c e ... (loop))))
```

10-5

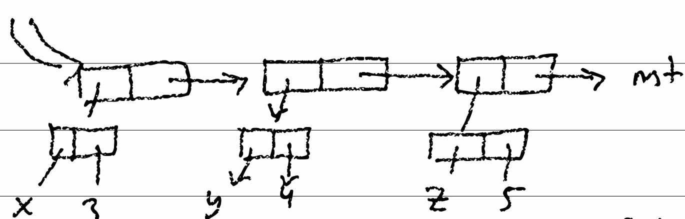
desugar (for [x := e_i; e_c; e_f]
e_b ...)
= (let ([x ~~box~~ e_i])
(while e_c
e_b ... e_f))

(let ([sum (box 0)])
(for [x := (box 0) ; (unbox x) ≤ 10 ; (set-box! x
(+1 (un x))]

(set-box! sum (+ (ub sum) (ub x)))
(unbox sum))

{ x: 3,
y: 4 } . y ⟹ (snd (pair 3 4))
x ⟹ fst , y ⟹ snd

{x: 3, y: 4, z: 5} ⟹ x ⟹ fst , y ⟹ fst snd , z ⟹ snd snd



mtobj = inl ##

field-set o f v =
cons (cons f v) o

11-1) (let ([sum (box 0)])
(for [x (box 0)]
(< (unbox x) 10)
(set-box! x (+ 1 (unbox x)))
(set-box! sum (+ (unbox x)
(unbox sum))))
(unbox sum))

$\Rightarrow$

$J_6 \to J_7$: e = ... | set! x e
E = ... | set! x E

(let ([sum 0])
(for [x 0] [< x 10) (set! x (+ 1 x))
(set! sum (+ sum x)))
sum)

(set! 1 2) "1 = 2" X
"f(1) = 2" X
"o.f = 2" ✓

(set! x e)

Lvalue = something with an address

11-24

OLD:

$$\Sigma \ / \ E[((\lambda\ (x)\ e)\ v)] \Rightarrow \Sigma \ / \ E[e[x \leftarrow v]]$$

NEW:

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

$$\emptyset \ / \ ((\lambda\ (x)\ (+\ x\ 7))\ 10) \overset{\text{OLD}}{\Longrightarrow} \emptyset \ / \ (+\ 10\ 7)$$

$$\overset{\text{NEW}}{\Longrightarrow} \emptyset[\sigma_1 \to 10] \ / \ (+\ (\text{unbox}\ \sigma_1)\ 7)$$

$$\Rightarrow \emptyset[\sigma_1 \to 10] \ / \ (+\ 10\ 7)$$

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

11-3

The machine implements J_6
(set-box!, but no set!)

The desugar transforms J_7 into J_6
(set!)

desugar x = (unbox x)
desugar (set! x e) = (set-box! x (desugar e))
desugar (λ (x₀) e) = if (set! x _) ∈ e
(λ (t₀) (let ([x₀ (box t₀)])
(desugar e))) o.w. (λ (x₀) (de e))

desugar
((λ (y) (set! y (+ y 1)) (+ y y)) 10) =
((λ (t₀) (let ([y (box t₀)])
(set-box! y (+ (unbox y) 1))
(+ (unbox y) (unbox y)))) 10)

((λ (x) (+ 1 x)) 10) ⇒
((λ (t) (let [(x (box t)]) (+ 1 (unbox x)))) 10)

desugar-top e = desugar (mutated-vars e)
e

letrec

11-4/ (~~let*~~ ([fac (λ rec (n)

(if (= n 0) 1

(* n (~~fac~~ rec (- n 1))))]

[even? (λ rec (n)

(if (= n 0) true

(odd? (- n 1))))]

[odd? (λ rec (n)

(if (= n 0) false

(even? (- n 1))))])

....)

11-5/ $(\text{let } () \; e) \overset{d}{\Rightarrow} e$

$(\text{let } ([x_0 \; e_0] \ldots [x_n \; e_n])$
$\quad e_b) \overset{d}{\Rightarrow} ((\lambda \; (x_0 \ldots x_n) \; e_b)$
$\qquad e_0 \ldots e_n)$

$(\text{let* } () \; e) \overset{d}{\Rightarrow} e$

$(\text{let* } ([x_0 \; e_0] \; [x_1 \; e_1] \ldots [x_n \; e_n]) \; e_b)$
$\overset{d}{\Rightarrow} (\text{let } ([x_0 \; e_0])$
$\quad (\text{let* } ([x_1 \; e_1] \ldots [x_n \; e_n]) \; e_b))$

$(\text{letrec } ([x_0 \; e_0] \ldots [x_n \; e_n]) \; e_b)$
$\overset{d}{\Rightarrow} (\text{let } ([x_0 \; \text{FALSE}] \ldots [x_n \; \text{FALSE}])$
$\quad (\text{set! } x_0 \; e_0) \ldots$
$\quad (\text{set! } x_n \; e_n)$
$\quad e_b)$

11-6

```
(let ([fac FALSE] [even? FALSE] [odd? FALSE])
    (set! fac (λ (n) (if (= n 0) 1
                          (* n (fac (- n 1))))))
    (set! even? (λ (n) (if (= n 0) TRUE
                            (odd? (- n 1)))))
    (set! odd? (λ (n) (if (= n 0) FALSE
                           (even? (- n 1)))))
    (even? (fac 5)))
```

```
(letrec ([f (λ (n) (g 0))]
         [x (f 5)]
         [g (λ (m) m)])
  x)   →  // fails w/ can't apply bool
```

Strategy 1: error-prone (Racket)

chang FALSE to UNDEFINED

Strategy 2: limiting (ML)

restrict RHS of letrec to fun defn (λ)

Strategy 3: hard to implement (Coq / Racket)

analyze the program and figure the problem

12-1) (/ 1 0) — δ is partial and undefined on δ(/, (1 0))

(5 3) — you can't all numbers as fns

(set-box! 7 0) — you can't set-box! numbers

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

→ (+ 2 (+ 2 2))

→ (+ 2 4) → 6

(5 3) $\not\mapsto$

```
cp = (V_NUM 3)   kp = (KApp [(VNUM 5)] [] ∅ k)
switch (cp -> tag ) // V_NUM
case V_NUM:
switch (kp -> tag) // K_APP
switch( kp -> vs [0] -> tag) { // V_NUM
   case V_PRIM: .... δ( ... ) ... ;
   case V_CLO : .... make new env, look at code ...;
   default: exit(1);
```

12-2

OLD: b = bool | num

NEW: b = bool | num | ~~[illegible]~~ err$_e$

OLD: E[(λ (x) e) v] → E[e[x ← v]]

E[p v] → E[δ(p, v)]

NEW: E[u v] → "Not a function" ∈ err$_e$

u ≠ p and u ≠ (λ (x) e)

an <u>abort</u>

E[X] → E[Y]

when ya throw away the context

J$_7$ → J$_8$: e = ... | abort e

E does <u>not</u> contain abort E

(+ 1 (+ 2 (abort (+ 3 (+ 4 0)))))

E = (+ 1 (+ 2 HOLE))

E[(abort (+ 3 (+ 4 0)))]

E[abort e] ⇒ e

CEK:

<abort e, env, k> ↦ <e, env, kret>

throw away

12-3/

```
int f (int x) {
    x += 8;
    return x;
    x *= 2;
    x++;
    return x+3;
}
f (5);
```

return is a "local" abort

k= (λ ()
x *= 2;
x ++
return x+3)

```
(define (fac n)
  (if (< n 0) (abort "Only ~~positive~~ non-neg ")
    (if (= n 0) 1
      (* n (fac (- n 1))))))
```

12-4/

```
(+ 1
   (try
      (+ 2
         (throw 3))        ⇏ 10
   catch                   ⇏ 6
   (λ (x) (+ x 4))))       ⇒ 8
```

$J_8 \Rightarrow J_9$: $e = \ldots \mid$ throw $e \mid$ try e catch e

$E = \ldots \mid$ try e catch E

$\mid$ try E catch v

$L = E$ except no try case

$E[\text{try } v_1 \text{ catch } v_2] \rightarrow E[v_1]$

$E[\text{try } L[\text{throw } e_1] \text{ catch } v_2] \rightarrow E[v_2\ e_1]$

$L[\text{throw } e_1] \rightarrow L[\text{abort } e_1] \rightarrow e_1$

$E =$ (+ 1 hole) $\quad L =$ (+ 2 hole)

$e_1 = 3 \qquad v_2 =$ (λ (x) (+ x 4))

$\rightarrow$ (+1 ((λ x. (+ x 4)) 3))

$\rightarrow$ (+1 (+ 3 4)) $\rightarrow$ (+1 7) $\rightarrow$ 8

12-5/ $CEK_4 \Rightarrow CEK_5$

$$
\begin{aligned}
K = \ldots \;&|\; \text{kif env } e \; e \; k \quad | \; \text{kapp } (v \ldots)\,(e \ldots) \text{ env } k \\
&|\; \text{kTryPre env } e \; k \quad | \; \text{kret} \\
&|\; \text{kTryPost } v \; k
\end{aligned}
$$

$\langle \text{try } e_1 \text{ catch } e_2,\ \text{env},\ k \rangle$

$\quad \mapsto \langle e_2,\ \text{env},\ \text{kTryPre env } e_1\ k \rangle$

$\langle v,\ -,\ \text{kTryPre env } e\ k \rangle$

$\quad \mapsto \langle e,\ \text{env},\ \text{kTryPost } v\ k \rangle$

$\langle v_1,\ -,\ \text{kTryPost } v_2\ k \rangle \mapsto \langle v_1,\ -,\ k \rangle$

$\langle \text{throw } e_1,\ \text{env},\ \text{kTryPost } v_1\ k \rangle$

$\quad \mapsto \langle e_1,\ \text{env},\ \text{kAPP } (v_1)\ ()\ -\ k \rangle$

$\langle \text{throw } e_1,\ \text{env},\ \text{kif env}'\ e_t\ e_f\ k \rangle$

$\quad \mapsto \langle \text{throw } e_1,\ \text{env},\ k \rangle$

$\langle \text{throw } e_1,\ \text{env},\ \text{kapp } (v \ldots)\ (e \ldots)\ \text{env}'\ k \rangle$

$\quad \mapsto \langle \text{throw } e_1,\ \text{env},\ k \rangle$

$\langle \text{throw } e_1,\ \text{env},\ \text{kTryPre env}'\ e\ k \rangle$

$\quad \mapsto \langle \text{throw } e_1,\ \text{env},\ k \rangle$

$\langle \text{throw } e_1,\ \text{env},\ \text{kret} \rangle \mapsto \langle e_1,\ \text{env},\ \text{kret} \rangle$

12-6

OLD: $(5 \quad 3) \not\rightarrow$

MID: $(5 \quad 3) \rightarrow$ error ("Not a fun")

NEW: $(5 \quad 3) \rightarrow$ throw error "Not a fun"

((λ (f) (f 3)) 5)

((λ (f)
  (try (f 3) catch (λ (x) 15)))
 5) ⇒ 15

$E[u \ v] \rightarrow E[\text{throw "Not a fun"}]$
  if $u \in P$ and $u \neq$ (λ (x) e)

(abort) $J_9 \Rightarrow$ try, throw

13-1 $J_8 \Rightarrow J_{10}$ cp

e = ... | call/cc e

```
E[call/cc v] ⇒ E[v (λ (x) (abort E[x]))]

(+ 1
   (call/cc (λ (esc)
              (let ([throw (λ (y) (esc (+ y 4)))])
                (+ 2 (throw 3))))))
⇒ E = (+ 1 ■)   // E[(call/cc J)]
(+ 1 ((λ (esc) ...) (λ (x) (abort (+ 1 x)))))
⇒ (+ 1 (+ 2 ((λ (y) ((λ (x) (abort (+ 1 x))) (+ y 4)))
              3)))
⇒ (+ 1 (+ 2 ((λ (x) (abort (+ 1 x))) 7)))
⇒ (+ 1 (+ 2 (abort (+ 1 7))))
⇒ (+ 1 7) ⇒ 8                        E[abort e] ↦ e
```

132/ desugar (try e_1 catch e_2) $\Rightarrow$

try-catch (λ () e_1) e_2

desugar (let/cc x e) $\Rightarrow$ callcc (λ (x) e)

standard library:

```
throw := (λ (x) ((unbox last-handler) x))
last-handler := (box (λ (x) (abort x)))
try-catch := (λ (body new-handler)
  (let ([old-handler (unbox last-handler)])
(begin0 (let/cc here (set-box! last-handler
                    (λ (x) (here (new-handler x))))
                 (body))
  (set-box! last-handler old-handler))))
```

13-3/ return statements:

```
(define (fac x)
  (when (< x 0) (return false))
  (if (= x 0) 1
      (* x (fac (- x 1)))))
=>

  desugar (define (f x ...) b)
    => (define (f x ...) (let/cc return b))
```

break/continue:

OLD

```
desugar (while c b) => ((λ loop () (when c b (loop))))
NEW    => ((λ loop ()
             (when c
               (let/cc break (let/cc continue b)
                             (loop)))))
```

13-4/ R^5RS (definition of Scheme) 1998

(much older)

"Programming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary."

$$CEK_4^{(abort)} \to CEK_6^{(call/cc)}$$

$$v = \dots \mid \mathrm{Kont}\ K$$

$$\langle v, -, \mathrm{kApp}\ [\mathrm{call/cc}]\ []\ -\ k \rangle$$

$$\mapsto \langle \mathrm{Kont}\ k, -, \mathrm{kApp}\ [v]\ []\ -\ k \rangle$$

$$\langle v, -, \mathrm{kApp}\ [\mathrm{Kont}\ k]\ []\ -\ - \rangle$$

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

13-5/ compiler from $J_{10} \xrightarrow{\text{(call/cc)}} J_8^{\text{(abort)}} / J_7^{\text{(ada)}}$

CPS - continuation-passing style

input:

(+ 1 (call/cc (λ (esc) (+ 2 (esc 3))))))

old: $+ : v \times v \to v$ call/cc : ~~4(...)~~

$(k \to v) \to v$

new: $+ : V \times V \times (V \to A) \to A$

~~let~~ call/cc : $((V \to A)^{\times (V \to A)} \to A) \times (V \to A) \to A$

call/cc := (λ (f k)
(f (λ (v nk) (k v)) k))

(call/cc (λ (esc k)
(esc 3 (λ (ans) (+ 2 ans k)))
(λ (ans) (+ 1 ans top)))

⇒ (esc 3 (λ (ans) (+ 2 ans k)))

[esc ~~↦~~ (λ (v nk) (+ 1 v top)),
k ↦ (λ (ans) (+ 1 ans top))]

⇒ (+ 1 3 top) ⇒ (top 4) ⇒ 4

13-6 ~~CPS-m := statements~~

$\cancel{a = b \mid x}$

$\cancel{call = (a \ \ldots)}$

$\cancel{v = b \mid (\lambda \ (x \ldots) \ call)}$

$\cancel{e = (\lambda \ (top) \ call)}$

$\cancel{\langle (\lambda \ (x \ldots) \ call), \ env \rangle}$

$\cancel{\mapsto \langle clo(}$

It's simple!

$$a = v \mid x$$

$$call = (a \ \ldots)$$

$$v = b \quad (\lambda \ (x \ldots) \ call)$$

i $$st = \langle call, \ env \rangle$$

$\langle (a_0 \ a_1 \ldots a_n), \ env \rangle$

$\mapsto \langle c, \ env' \rangle$

where $\alpha(a_0) = (\lambda \ (x_1 \ldots x_n) \ c)$

$v_1 \ldots v_n = \text{map} \ \alpha \ (a_1 \ldots a_n)$

$env' = \emptyset[x_1 \mapsto v_1] \ldots [x_n \mapsto v_n]$

$\alpha \ b = b \qquad \alpha \ x = env[x]$