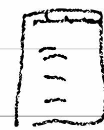
1-1

Goal: Implement a VM

What does a program mean?

"1+1" means "2"

"wordc" means 

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

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

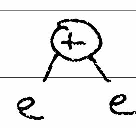
BNF or CFG

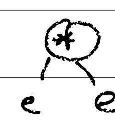
J_0: $e ::= v \mid (+\ e\ e) \mid (*\ e\ e)$

$v ::= \text{number}$

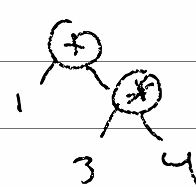
"(+ 1 (* 3 4))" $\in J_0.e$

"(+ 1))"

$e ::= v$ or 

or 

$v ::= \text{number}$



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) : $e \to v$

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* ≈ V*

E*

1-3/

```
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

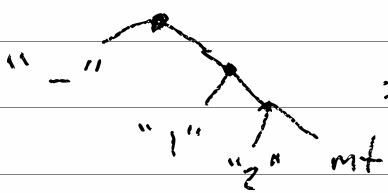
string expr

S-expr

Sexpr

$\Longrightarrow J_0$

Sexpr = string or pair of Sexpr
or empty

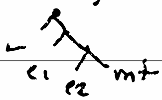


= (- 1 2)

1 - 2

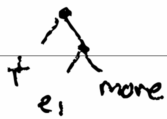
2-1/ Sexpr $\quad$ J_0

desugar (- e_1) = (* -1 (desugar e_1))



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

(+) = 0



(+ e_1 . more) = (+ (d e_1) (d (+ . more)))

(*) = 1

(* e_1 . more) = (* (d e_1) (d (* . 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) =

$\quad$ c = interp e_c

$\quad$ e_k = <u>if</u> c e_t <u>o.w</u> e_f

$\quad$ return interp e_k

interp (e_p e_0 ... e_n) =

$\quad$ p = interp e_p (must be a prim)

$\quad$ v_0 ... v_n = interp v_0 ... interp v_n

$\quad$ ret $\delta(p, v_0 \ldots v_n)$

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-likee wrt 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 (e_0 ... e_i e_{i+1} ... e_n)
= (e_0 ... e_i' e_{i+1} ... e_n)

2-3)

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

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

A context, C ::= hole
| if C e e
| if e C e
| if e e C
| (e ... C e ...)

plug : C x e ⇒ e

plug hole e = e (plug C ep)

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

plug (e_b... C e_a...) e_p = (e_b... (plug C e_p) e_a...)

plug (■ + (2+3)) (1+1) = (1+1) + (2+3)

C 2 = 2 + (2+3)

Parse : e ⇒ C x e

2-4

$$e \Rightarrow e$$

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

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

step $C[p \ v0 \ldots vn] = C[\delta(p, v0 \ldots vn)]$

find_redex : $e \Rightarrow C \times e$

↑ redex

reducible expression

When do two programs mean the same thing?

"1 + 2" "2+1"

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

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 × secs

3-1 step : $e \Rightarrow e$ $\longrightarrow\longrightarrow\longrightarrow$ v

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

.... + ... * ... − .... (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

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

$E[\text{if true } e_t\ e_f] \to E[e_t]$
$E\{\text{if false } e_t\ e_f] \Rightarrow E[e_f]$
$E[(p\ v_0 \ldots v_n)] \to E[\delta(p, v_0 \ldots v_n)]$

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) =

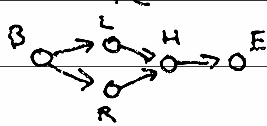
  (E, e_0') = fr e_0

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

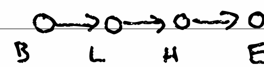
  new EApp(v, E, e_1 .. e_n)

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

$step_C$

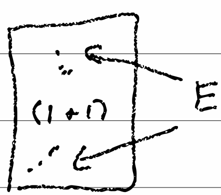


$step_E$



The Standard Reduction theorem

Alonzo Church   ((Church-Rosser)   Rosser



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

  ..... (if .... hole ...)$^{(+34)}$ ...))

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

3-3) $\text{step}_E : e \Rightarrow e$

machine semantics

$$
\begin{tikzcd}
e \arrow[rr, "\text{step}_e"] \arrow[d, "\text{inject}"'] & & e' \arrow[r, "s"] & e'' \arrow[r] & e''' \arrow[r] & v \\
st \arrow[rr, "\text{step}_{st}"'] & & st' \arrow[u, "\text{extract}"'] \arrow[r] & \arrow[r] & st''' \arrow[u, "\text{extract}"'] \arrow[r] & st'''' \arrow[u, "\text{extract}"']
\end{tikzcd}
$$

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

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

$\text{extract } \langle e, E \rangle = E[e]$

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

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

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

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 }
    cc0-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

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

kontinuation $kapp\ (v\ldots)\ (e\ldots)\ k$ // (v... E e...)

stac k

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

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

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)))  ] = Δ
(define (g x) (f x x))                 ]
(+ 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)

$\langle \Delta, v_n, kapp((f\ v_0 \ldots), ()), k \rangle$ N

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

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

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

(if (x > 10 mil) (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) $CK_1 \to CEK_0$

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

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

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

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

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

$\text{done? } \langle \Delta, v, env, kret \rangle$

!!! ↓ WRONG ↓ !!! (look ahead 2 pgs)

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

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

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

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

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

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

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

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

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

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

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

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

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

6-3) (define (f x) (+ x y))

(define (g y) (f 5))

(g 10)

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

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

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

$\mapsto \langle (+\ 5\ y),\ \text{''} \rangle$

$\mapsto \langle (+\ 5\ 10),\ \text{''} \rangle$

$\mapsto \langle 15,\ \text{''} \rangle$

WRONG:

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

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

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

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

(define (g y) ~~true~~)

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

(f 6)

Dynamic Scope

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

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

RIGHT

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

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

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

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

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

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

$$\langle \Delta, v_n, \_\_, \mathrm{kapp}((p\ v_0 \ldots), (), \_\_, k) \rangle$$
$$\mapsto \langle \Delta, \delta(p, (v_0 \ldots v_n)), mt, k \rangle$$

$$\langle \Delta, v_n, \_\_, \mathrm{kapp}((f\ v_0 \ldots), (), \_\_, k) \rangle$$
$$\mapsto \langle \Delta, e, mt[x_0 \leftarrow v_0] \ldots [x_n \leftarrow v_n], k \rangle$$

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

6-5/ $J_2 \rightarrow J_3$

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

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

$x :=$ variable names

$b :=$ number | bool | 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

new

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

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

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

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

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

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

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

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

old ↑ $\quad\quad$ $\langle 5, \text{mt}, \text{kret}\rangle$

$\to 5$

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

CEK

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

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

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

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

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

6-7/

desugar (let ([x_0 e_0] .... [x_n e_n]) be)
= ((λ (x_0 ... x_n) be) e_0 ... e_n)

```
(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)

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

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

⇒ true M N ⇒ $(\lambda x y. x)$ M N ⇒ M

7-2/ Church-encoded numbers

numbers are iteration

num : Thing To Do $\Rightarrow$ Something to do it to

$\Rightarrow$ 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/ $\text{zero?} = \lambda n.\ n\ (\lambda x.\ \text{false})\ \text{true}$

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

$\Rightarrow T$

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

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

$\Rightarrow F$

$\text{fst}\ (\text{pair}\ M\ N) \Rightarrow M$

$\text{snd}\ (\text{pair}\ M\ N) \Rightarrow N$

$\text{pair} := \lambda x. \lambda y. \lambda \text{sel}.\ \text{sel}\ x\ y$

$\text{fst} := \lambda p.\ p\ \text{true}$

$\text{snd} := \lambda p.\ p\ \text{false}$

$\text{fst}\ (\text{pair}\ M\ N) \Rightarrow (\text{pair}\ M\ N)\ \text{true}$

$\Rightarrow (\text{true}\ M\ N) \Rightarrow M$

$\text{select}\ (\text{inl}\ M)\ f\ g \Rightarrow f\ M$

$\text{select}\ (\text{inr}\ N)\ f\ g \Rightarrow g\ N$

$\text{inl} := \lambda v. \lambda f. \lambda g.\ f\ v$

$\text{inr} := \lambda v. \lambda f. \lambda g.\ g\ v$

$\text{select} := \lambda o. \lambda f. \lambda g.\ o\ f\ g$

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)

one

copy, not a reference

(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)

$\mathcal{Z}$-5/ $\Omega := \omega \quad \omega$

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

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

$(\lambda x.\ (x \quad x))\ \ (\lambda x.\ (x\ \ x))$

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

$\Rightarrow \quad \omega \qquad\qquad \omega$

$\Rightarrow \quad \Omega$

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
~~[illegible]~~ (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:

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

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

NEW:

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

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

$CEK_1 \Rightarrow CEK_2$

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

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

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

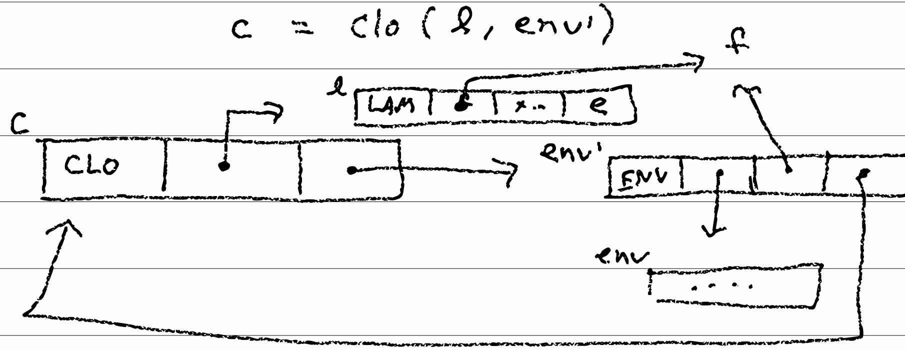
where $\ell = (\lambda\ (x_0 \dots x_n)\ e)$

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

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

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

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



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)

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

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

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

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 ~~#false~~ v

inr v = pair ~~#true~~ v

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

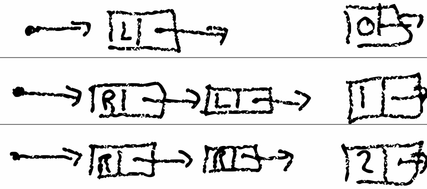
(let ([vs es]) (if (fst vs) (let ([x_1 (snd vs)]) e_1)
(let ([x_2 (snd vs)]) e_2)))

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

○ = inl ....

□ = inr (inl ...)

△ = inr (inr ...)



Algebraic Data Types

Type = 1 // unit ++

0 // Nothing

Type × Type // Pair (pair e1 e2)

Type + Type // Variant (inl e1)(inr e2)

B // Base types 5

B = Int32 | ...

Bool = 1 + 1 Nat = 1 + Nat 110

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

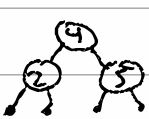
false = inr ++ 10 one = inr (inl ++)

List A = 1 + (A × List A)

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

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

Tree A = 1 + (Tree A × A × Tree A)



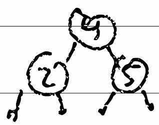
= inr (pair (inr (pair (inl ++) (pair 2 (inl ++))))
(pair 4 (inr (pair (inl ++) (pair 5 (inl ++))))))

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

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 $\nearrow$ sum $\nearrow$

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 $\Rightarrow : e \to e$ $\quad E[$(if true e_t e_f)$] \Rightarrow E[e_t]$

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

$\Sigma : \sigma \to v$

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

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

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

$\Sigma \times E[\text{set-box! } \sigma \ v] \Rightarrow \Sigma[\sigma \mapsto v] \times E[\overset{v}{tt}]$ not C way

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

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

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

$\langle \text{if } e_c\ e_t\ e_f, env, sto, k \rangle \mapsto \langle e_c, env, sto, \text{ifk } env\ e_t\ e_f\ k \rangle$

$\langle v, -, sto, \text{appk } (box)\ ()\ -\ k \rangle$

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

option 2: box is a primitive that does

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

(unbox (box v)) = ret ptr

(set-box! [V_BOX, ptr1] ptr2) = changes to memory to

[V_BOX, ptr2]

```
obj_t* delta_setbox ( obj_t* args ) {
((box_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))

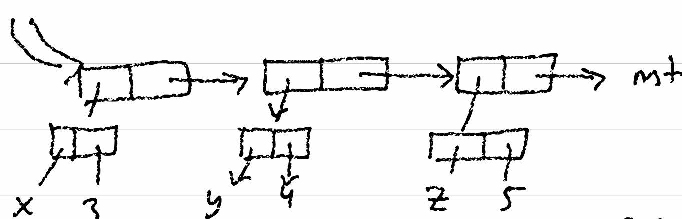
(let ([sum (box 0)])
(for [x := (box 0); (unbox x) ≤ 10; (set-box! x
(+1 (un x)))]

(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 tt

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

(set! x e)

$\nearrow$ Lvalue = something with an address

"o.f = 2" ✓

11-24

OLD:

$$\Sigma \;/\; E[((\lambda\ (x)\ e)\ v)] \Longrightarrow$$
$$\Sigma \;/\; E[\,e[x \leftarrow v]\,]$$

NEW:

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

$$\emptyset \,/\, ((\lambda\ (x)\ (+\ x\ 7))\ 10) \xRightarrow{\text{OLD}} \emptyset \;/\; (+\ 10\ 7)$$
$$\xRightarrow{\text{NEW}} \emptyset[\sigma_1 \to 10] \;/\; (+\ (\text{unbox}\ \sigma_1)\ 7)$$
$$\Rightarrow \emptyset[\sigma_1 \to 10] \;/\; (+\ 10\ 7)$$

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

11-3

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

The desugar transforms J_7 (set!) into J_6

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

11-4/

```
(~~let*~~ letrec ([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/ (let () e) $\xRightarrow{d}$ e

(let ([x_0 e_0] ... [x_n e_n])
    e_b) $\xRightarrow{d}$ ((λ (x_0 ... x_n) e_b)
        e_0 ... e_n)

(let* () e) $\xRightarrow{d}$ e

(let* ([x_0 e_0] [x_1 e_1] ... [x_n e_n]) e_b)
$\xRightarrow{d}$ (let ([x_0 e_0])
    (let* ([x_1 e_1] ... [x_n e_n]) e_b))

(letrec ([x_0 e_0] ... [x_n e_n]) e_b)
$\xRightarrow{d}$ (let ([x_0 FALSE] ... [x_n FALSE])
    (set! x_0 e_0) ...
    (set! x_n e_n)
    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: errorprone (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) $\not\to$

cp = (V_NUM 3) kp = (KApp [(VNUM 5)] [] ø 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, lookatcode ...;

default: exit(1);

12-2

OLD: $b = \text{bool} \mid \text{num}$

NEW: $b = \text{bool} \mid \text{num} \mid \text{err}_e$

OLD: $E[(\lambda\ (x)\ e)\ v] \to E[e[x \leftarrow v]]$

$E[p\ v] \to E[\delta(p, v)]$

NEW: $E[u\ v] \to$ "Not a function" $\in \text{err}_e$

$u \neq p$ and $u \neq (\lambda\ (x)\ e)$

↑ an <u>abort</u>

$E[X] \to E[Y]$ when ya throw away the context

$J_7 \to J_8$: $e = \ldots \mid \text{abort}\ e$

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

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

E = (+ 1 (+ 2 HOLE))

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

$E[\text{abort}\ e] \Rightarrow e$

CEK:

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

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)) $\cancel{\Rightarrow}$ 10
catch $\cancel{\Rightarrow}$ 6
(λ (x) (+ x 4)))) $\Rightarrow$ 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) $L =$ (+ 2 hole)
$e_1 = 3$ $v_2 =$ (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 3) $\not\to$

MID: (5 3) → erre ("Not a fun")

NEW: (5 3) → throw erre "Not a fun"

((λ (f) (f 3)) 5)

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

$E[u\ v] \to E[\text{throw "Not a fun"}]$
if $u \notin P$ and $u \neq (\lambda\ (x)\ e)$

```
          (abort)                    J9 ⇒ try, throw
13-1      J8 ⇒ J10                        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
```

13.2/ desugar (try e_1 catch e_2) ⇒

try-catch (λ () e_1) e_2

desugar (let/cc x e) ⇒ 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."

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

$$v = \ldots \mid \text{Kont } K$$

$$\langle v, \_, \text{kApp } [\text{call/cc}]\ [\,]\ \_\ k \rangle$$
$$\mapsto \langle \text{Kont } k, \_, \text{kApp } [v]\ [\,]\ \_\ k \rangle$$
$$\langle v, \_, \text{kApp } [\text{Kont } k]\ [\,]\ \_\ \_ \rangle$$
$$\mapsto \langle v, \emptyset, k \rangle$$

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

CPS - continuation-passing style

input:

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

old: $+ : v \times v \Rightarrow v$ call/cc : ~~...~~

$(k \Rightarrow v) \Rightarrow v$

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

call/cc : $((V \Rightarrow A) \xrightarrow{\times (V \Rightarrow A)} \Rightarrow A) \times (V \Rightarrow A) \Rightarrow A$

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~~

$$a = b \mid x$$
$$call = (a \ \ldots)$$
$$v = b \mid (\lambda \ (x \ldots) \ call)$$
$$e = (\lambda \ (top) \ call)$$

$$\langle (\lambda \ (x \ldots) \ call), env \rangle$$
$$\mapsto \langle clo($$

It's simple!

$$a = v \mid x$$
$$call = (a \ \ldots)$$
$$v = b \mid (\lambda \ (x \ldots) \ call)$$
$$; \quad st = \langle call, env \rangle$$

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

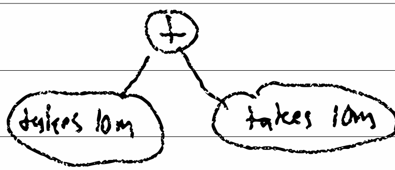
where $\alpha(a_0) = (\lambda \ (x_1 \ldots x_n) \ c)$

$$v_1 \ldots v_n = map \ \alpha \ (a_1 \ldots a_n)$$
$$env' = \emptyset [x_1 \mapsto v_1] \ldots [x_n \mapsto v_n]$$

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

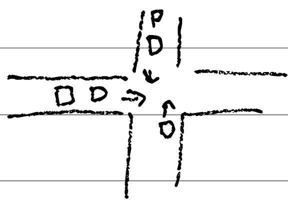
15-1) Concurrency — logical simultaneity

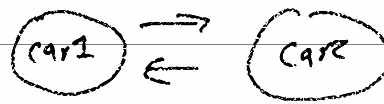
Parallelism — concrete simultaneity



Serial / Sequential - 20m

Parallel — 10m





A X
B Y

possible programs:

A B X Y — 1→2

X Y A B — 2→1

A X B Y — interleave 1st

— 2st

X A Y B

X A B Y — car2 / car1

A X Y B — car1 / car2

A Y X B X

Concurrent has N ans

Parallel has 1 ans

15-2/

```
spawn! : (-> a) -> void
exit!  : () -> void
```

< = happens before

```
(begin (spawn! (λ () (print 'A)
                     (print 'B)))
       (spawn! (λ () (print 'X)
                     (print 'Y)))
       (exit!))
```

A < B
X < Y

```
(define RQ empty)   ;; RQ = ready queue
(define (spawn! t)
  (set! RQ (cons (λ () (begin (t) (exit!))) RQ)))
(define (exit!)
  (if (empty? RQ) (void)
      (let ([next (first RQ)])
        (set! RQ (rest RQ))
        (next))))
```

15-3/ (let ([m 10])
  (spawn! (λ () (set! m (* 2 m))))
  (spawn! (λ () (set! m (+ 2 m))))
  (exit!))

$10 \xrightarrow{\times} 20 \xrightarrow{+} 22$ $\qquad$ $10 \xrightarrow{+} 12 \xrightarrow{\times} 24$

make-channel : () → Chan
send! : Chan × A → ()
recv! : Chan → A

A₂ (send! ch 0)

(let ([ch (make-channel)])
  (spawn! (λ () (send! ch 20))) A₁
  (spawn! (λ () (print (+ 2 (recv! ch))))) B
  (exit!))

$\boxed{AB} = 22$ $\quad$ $\boxed{BA}$ = dead $\quad$ "match pair"

(spawn! (λ () (print (- (recv! ch) 2))))

ABC = 22 $\quad$ BAC = 18 $\quad$ CAB = 22
ACB = 18 $\quad$ BCA = 0 $\quad$ CBA = 0

~~A~~ 22, 8 — B, C
18, 20 — C, B

15-4

make-chan : () → Chan A
send! : Chan A × A → ()
recv! : Chan A → A

Empty Channel — SendCh

Chan A = Box (1 + (A × (() → B) × Chan A)
+ ((A → C) × Chan A))

RECV ch

```
make-chan := (box tt)
send! ch v :=
    case (unbox ch) of
        RecvCh f ch' → (spawn! (λ () (f v)))
                        (set-box! ch ch')
        next → (let/cc me
                   (set-box! ch (SendCh v me next))
                   (exit!))
recv! ch :=
    case (unbox ch) of
        SendCh v f ch' → (spawn! f)
                          (set-box! ch ch'); v
        next → (let/cc me
                   (set-box! ch (RecvCh me next))
                   (exit!))
```

15-5/ make-promise : $(\Rightarrow A) \to$ Promise A
fulfill : Promise $A \to A$

```
make-promise f := (let ([ch (make-chan)])
                    (spawn! (λ() (send! ch (f))))
                    ch)
fulfill pch := recv! pch
```

16-1 errors like (+ 1 #true)
resulted in (abort "can't add bools")

#1 → 2 (C logic) JS
#2 → crash i.e. abort (~~Python~~ logic)
#3 → exception (Python logic)

#1: UNSOUND #2: UNSAFE
#3: SAFE

p := .... | number? | box? | bool? | prim? | pair? | ...
| function-arity

((λ (x) (+ 1 x)) 1 2)

+

```
(define (safe+ x y)
  (if (and (number? x) (number? y))
      (+ x y)          unsafe+ = #%+
      (throw "Not a number")))
```

16-2

```
(define (f x) (* x 2))
(f 1 2)

desugar (e0 ... en) = (SAFE-CALL e0
                          (list e1 ... en))

(define (f x) (SAFE-APPLY * (list x 2)))
(SAFE-APPLY f (list 1 2))


(define (SAFE-APPLY f args)
  (if (function? f)
      (if (= (length args) (function-arity f))
          (#%apply f args)
          (throw "wrong num")) (throw "not fun")))
```

$$\langle v_e, \_, \text{kAPP}([\#\%\text{apply}\ v_0]\ []\ \_\ k) \rangle$$

$$\longmapsto \langle v_N, \_, \text{kApp}\ [v_0\ v_1 \ldots v_{n-1}]\ []\ \_\ k \rangle$$

where $v_e = (\text{list}\ v_1 \ldots v_n)$

(apply + (list 2 3)) ⇒ (+ 2 3) ⇒ 5

16-3/ bst-insert : BST? × Num? → BST

```
(define (bst-insert b v)
  (unless (BST? b) (throw ...))
  (unless (num? v) (throw ...))
  ....)
```

MOD :

unsafe-f_1 : .... unsafe-f_2 ....
f_2
f_3

f_1 : Check (unsafe-f_1 ...)
f_2 : check (unsafe-f_2 ...)

protect
+ −
apply-contract : Value × Contract × Blame × Blame

```
(define bst-insert
  (apply-contract unsafe-bst-insert
                  (fun-ctc (list BST? num?)
                           BST?)
                  "Me" "Them"))
```

16-4) desugar (module
(with-module
body ...
#: exports
[f ctc] ...) user ...)

⇒ (apply (λ (f ...) user ...)
(letrec (body ....)
(list (contract f ctc ME THEM)
...)))

Contract = FlatCtc (A → Bool)
FunCtc (List contract) contract

BST? = FlatCtc (λ (x) ....)

protect v (FlatCtc pred) pos neg =
(if (pred v) v (error "pos" "is at fault"))

protect v (FunCtc dom rng) pos neg =
(if (! (function? v) (= (length dom) (fun-arity f)))
(error pos " not a fun or wrong arity ")
(λ args
(if (! (= (len args) (len dom)) (error neg))
(protect (apply v (map (λ (a ctc) (protect a ctc neg pos))
any dom))
rng pos neg)))))

16-5/

(protect map
(+) (FunCtc (−) (list (FunCtc (+) (list Num?)
(−) Bool?)
(ListCtc (+) Num?)
(+) (ListCtc Bool?)))

17-1 Syntactic Extension

(time e) ⇒ return value of e and print how long it took to run

now ∈ p

$E[(\lambda x. b)\ v] \to E[b[x \leftarrow v]]$

```
(define (time' x)
  (let* ([before (now)]
         [ans (x)]
         [after (now)])
    (displayln (- after before))
    ans))
```

~~(time (fib 100))~~

(time' (λ () (fib 100)))

C] cpp

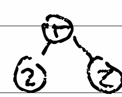
```
#define TIME(e) (time' (λ () e))
```

C macros operate on text of programs NOT values of the program

17-2/ cpp problems

```
#define SUB -␣2
...
return x SUB;   =>   x - 2
...
```

```
#define FOO 2);
...
printf("The number is %d", FOO
return 1;
```

"(+ 2 2)" ⟺ 

text editor — program tools (compiler/vm)

17-3/ macros : syntax → syntax (like desugar)

macros-by-example (1986) – we'll implement

(define-syntax-rules let ← macro name
  [(let ([x xe] ...) be ...) ← pattern
   ((λ (x ...) be ...) xe ...)]) ← template

(dsr name [pat template] ...)

expand : SetofRules × ~~SExpr~~ Sexpr → ~~S~~ J

(let ([x 3] [y 4]) (+ x y))

expand Σ (cons mac rest) =
  if Σ(mac) = defn then
    ~~exp~~ expand1 defn (cons mac rest)
  else
    App (expand Σ mac , map (expand Σ) rest)

expand Σ (Num n) = (Num n)

expand Σ (cons 'lambda (cons args body)) =
  Lambda args (expand Σ body)

17-4 expand 1

pat_o

defn = (list (pair (let ([x xe] ...) be ...) $temp_o$
((λ (x ...) be ...) xe ...)))

use = (let ([a 3] [b 4]) (+ a b))

match pat_o use =
[x ↦ (a b)
xe ↦ (3 4)
be ↦ ((+ a b))] = σ

match : Pat × Sexpr ⇒ Env or F

transcribe $temp_o$ σ =
((λ (a b) (+ a b)) 3 4)

expand d Σ (define-syntax-rules mac defns)
= Σ[mac] = defns

17-5

match : Pat × Sexpr → Env or false

match () () = ∅

match b b = ∅

match x se = [x ↦ (0, se)]

match (p '...) se =

merge (map (match p) se)

match (cons pl pr) (cons l r) =

match pl l ∪ match pr r

merge : List (Env or false) → Env

merge [] = ∅

merge [.... false ...] = false

merge [σ_0 σ_1 ...] =

new hash where k ... = keys of σ_0

k ↦ (level+1, vs)

where $\sigma_0(k)$ = (level, _)

vs = (list snd($\sigma_0(k)$) snd($\sigma_1(k)$) ...)

17-6

match ([x xe] ...)
    ([a 3] [b 4]) =

merge (map (match [x xe])
    (list [a 3] [b 4])) =

merge (list (match [x xe] [a 3])
    (match [x xe] [b 4])) =

merge (list [$x \mapsto (0, a)$, $xe \mapsto (0, 3)$]
    [$x \mapsto (0, b)$, $xe \mapsto (0, 4)$]) =

[$x \mapsto (1, (a\ b))$
  $xe \mapsto (1, (3\ 4))$]

~~17~~ 7/ transcribe : env x template ⇒ sexpre

transcribe σ ( ) = ( )
transcribe σ b = b
transcribee σ x = se   where σ(x) = (0, se)
tr σ (p ...) =
map ~~[illegible]~~ (λ (σ') (tr σ' p))
(decompose σ)
tr σ (cons ~~pa~~ ~~pd~~ ta td) = (cons (tr σ ta)
(tr σ td))

decompose : env → list env
decompose σ = (list σ₀ ... σn)
where n = length of (cdr σ[x]) for some x
σᵢ[x] = (level − 1 , vs[i])
where (level, vs) = σ[x]

17-8/
```
(define-syntax-rules or
  [(or)  true]
  [(or x  y ...)
   (let ([tmp x])
     (if tmp tmp (or y ...)))])
```

```
(or false 3 4) => 3
          ↗
  (begin (display "Hi!")
         3)
```

```
(let ([tmp_red 7])
  (or false tmp_red)) => 7
(let ([tmp_red 7])
 (let ([tmp_bl false])
  (if tmp_bl tmp_bl
      (let ([tmp_gr tmp_red])
        (if tmp_gr tmp_gr false))))) =>
```

macro : stx → stx

18-1/ Backtracking Non-determinism

```
nde  =  (ans v)              : Ans
     |  fail                 : ⊥
     |  (choice nde nde)     : (ND A) x ND A → ND A
     |  (bind nde f)         : ND A → (A → ND B) →
                                 ND B
```

run : ND a → ~~list a~~ stream a

```
empty-stream : empty
stream-cons  : a × (→ stream a) → stream a
stream-first : stream a → a
stream-rest  : stream a → stream a
```

```
run p = sols (list (st p (kont:return)))
sols '() = empty-stream
sols (cons (st p k) g) =
 case p of (bind p' f) → sols (cons (st p' (kont:bind f k))
                                                          g)
     (choice p1 p2) → sols (cons (st p1 k) (st p2 k) g)
     (fail) → sols g
     ans v → case k of  return → stream-cons v
                                        (λ () (sols g))
                 bind f k → sols (cons (st (f v) k) g)
```

18-2/ query : Question → ND ans
query q = bind (searchN DB ∅ (list q))
(λ (env) (ans (transcribe env q)))

searchN : Rules × Env × List(Questions) → ND Env
searchN rules env [] = ans env
q : qs = bind (searchN rules env qs)
(λ (env') (search* rules env' rules q))

search* : Rules × Env × Rules × Question → ND Env
search* all env some q = case some of
[] → fail | ruleN : rules →
choice (search* all env rules q)
(search1 all env ruleN q)

search1 : Rules × Env × Rule × Question → ND Env
search1 all env (conclusion, deps) q =
(bind (match conclusion q)
(λ (env') (searchN all ~~env~~ env' deps)))

18-3/ Non-determinism Expressions are a Monad

Monad A =

    return : A → Monad A
    bind : Monad A → (A → Monad B)
                → Monad B

List Monad
    return x = [x]
    bind mx f = map f mx

bind (return 1) add1 ⇒ [2]
bind (list 1 2 3) add1 ⇒ [2 3 4]
bind x (λ (a)
    bind ~~return~~ (list (add1 a) (sub1 a))
        mult2)) ⇒

19-1 / Memory Management

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

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

$k = \mathrm{ret} \mid \mathrm{kfun}(e, env, k) \mid \mathrm{karg}(v, k)$

$\langle x, env, k\rangle \mapsto \langle env[x], \emptyset, k\rangle$

$\langle \lambda x.e, env, k\rangle \mapsto \langle \mathrm{clo}(\lambda x.e, env), \emptyset, k\rangle$

$\langle e_1\ e_2, env, k\rangle \mapsto \langle e_1, env, \mathrm{kfun}(e_2, env, k)\rangle$

$\langle v, \emptyset, \mathrm{kfun}(e_2, env, k)\rangle \mapsto \langle e_2, env, \mathrm{karg}(v, k)\rangle$

$\langle v, \emptyset, \mathrm{karg}(\mathrm{clo}(\lambda x.e, env), k)\rangle \mapsto \langle e, env[x \mapsto v], k\rangle$

If one object has two pointers to it, we say it is "aliased"

MM = how to manage scarce memory
- how to allocate — malloc impl
- when to free — you call free

right ~~wrong~~ oom ~~crash~~

soundness =

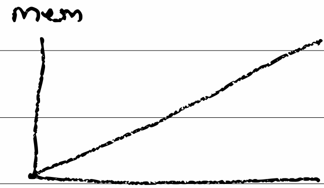
efficiency $\begin{cases}\text{space}\\ \text{time}\end{cases}$

performance

19-2 soundness — right ~~wrong~~ OOM ~~crash~~

efficiency

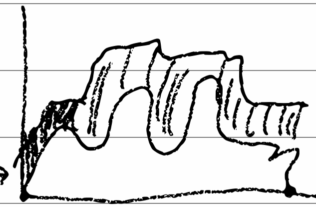
— space



infinite mem

peak usage = highest point

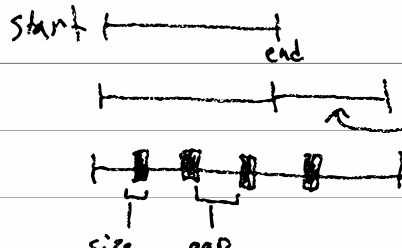
diff is space$_{eff}$



actual

perfect

— time



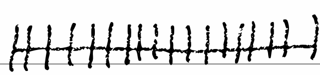
perfect

more time for MM

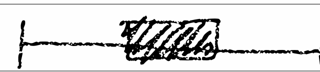
performance

size = latency

gap = period

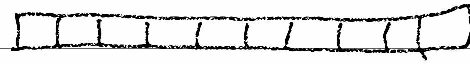


low latency



high latency

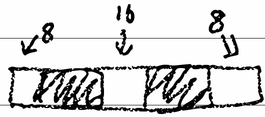
14-3/ C-style malloc & free

mem = 

malloc(sizeof(node)) = give me 16 spots

free(ptr) = figure out adj free space

O(lg n)



= fragmentation

time eff = malloc = free = O(lg n)

space eff = O(lg n)

soundness = not

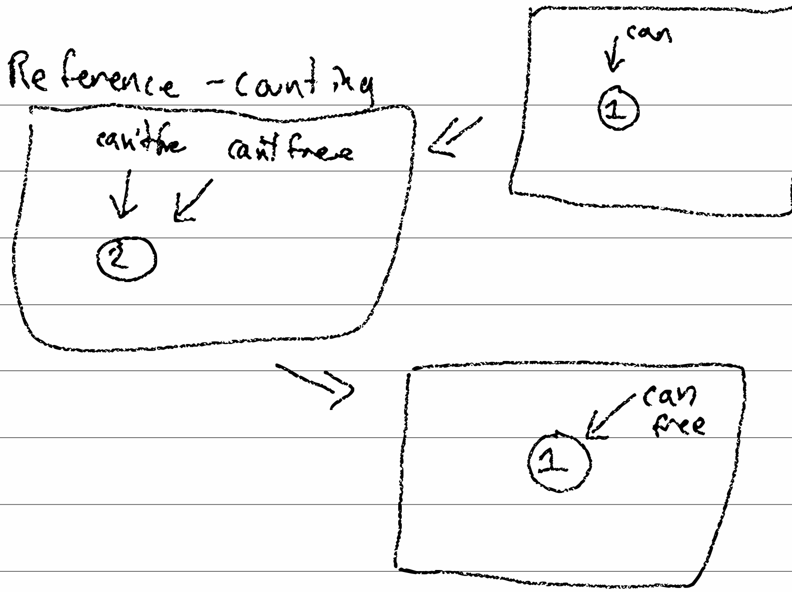
performance = •

depend on programs

soundly:

turn all aliases into copies (half copying, X2 free)

19-4/ Reference-counting

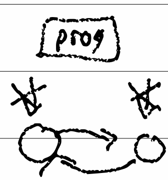


malloc: ref=1 ref: ref++ unref: ref--
if (ref==0) {free}

sound: who calls ref/unref? if programmer ⇒ unsound
if lang ⇒ sound

space eff: every object stores count
LL (16) ⇒ (16 + CNT) CNT = [1, 8]

prog



cycles are never deleted

time eff: ∝ work of program $O(w)$ not $O(n)$ mem

latency: cascading deletes of large frees = high latency
manage w/ queues

20-1 Garbage Collection (a sound memory manager)

John McCarthy for LISP in 1959

Mark & Sweep

....

var x = ...;

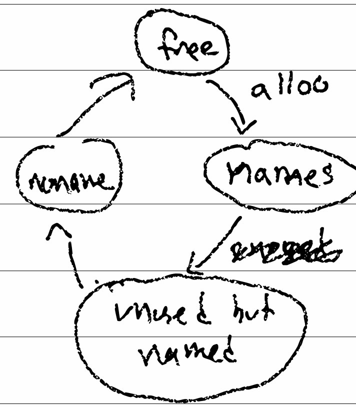
line 20: .... x .... x ... x ...

if (cc?) {

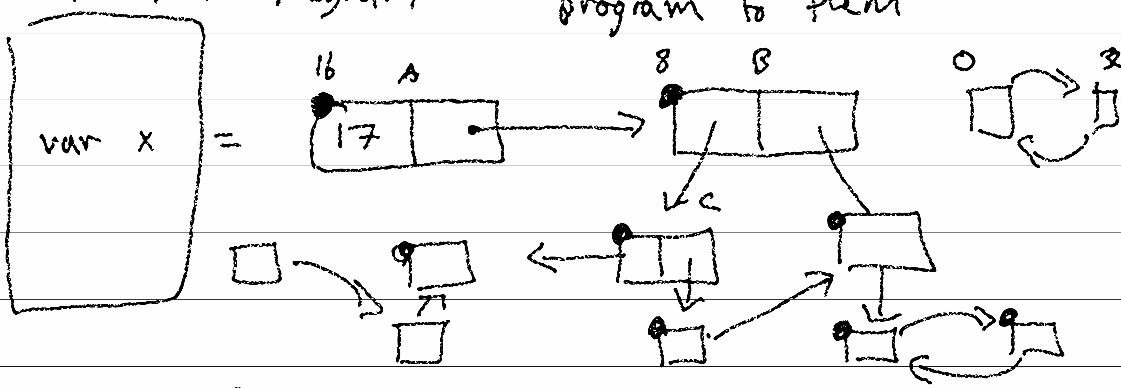
line 42: ... x ...}

[never see x again

return;



the working set = all objects with paths from rootset = program program to them



c= x[1][0]

garbage = mem - working

10-21 mark = from root set, mark all children recursively

sweep = go through mem, delete if no mark

remove mark o.w.

CESK+MS : $st = \langle \sigma, \sigma, \sigma, \Sigma \rangle$ $\quad \sigma = \emptyset \mid n$

$\Sigma = \cancel{\emptyset} \; \Sigma[\sigma \mapsto *]$ $\quad k = \text{kret}$

$e = v \mid @\, \sigma \; \sigma \mid x$ $\quad \mid \text{kfun}(\sigma, \sigma_{k})$

$v = b \mid \boxed{\lambda x.\, \sigma} \mid \text{clo}(\sigma, \sigma) \mid \text{karg}(\sigma, \sigma)$

$env = \cancel{\emptyset} \; \sigma[x \mapsto \sigma]$

$\langle \sigma_e, \sigma_{env}, \sigma_k, \Sigma \rangle = \text{case } \Sigma(\sigma_e) \text{ of}$

$x \to \langle \Sigma(\sigma_{env})[x], \emptyset, \sigma_k, \Sigma \rangle$

$\lambda x.\sigma_b \to \langle \sigma_{clo}, \emptyset, \sigma_k, \Sigma' \rangle \quad \Sigma' = \Sigma[\sigma_{clo} \mapsto \text{clo}(\sigma_e, \sigma_{env})]$

$@\, \sigma_L \; \sigma_R \to \langle \sigma_L, \sigma_{env}, \sigma_{k'}, \Sigma' \rangle \quad \Sigma' = \Sigma[\sigma_{k'} \mapsto \text{kfun}(\sigma_R, \sigma_{env}, \sigma_k)]$

$\_ \to \text{case } \Sigma(\sigma_k) \text{ of}$

$\text{kfun}(\sigma_R, \sigma_{env}, \sigma_{k'}) \to \langle \sigma_R, \sigma_{env}, \sigma_k, \Sigma' \rangle \quad \Sigma' = \Sigma[\sigma_k \mapsto \text{karg}(\sigma_e, \sigma_{k'})]$

$\text{karg}(\sigma_{clo}, \sigma_{k'}) \to \langle \sigma_{body}, \sigma_{env''}, \sigma_{k'}, \Sigma' \rangle$

where $\Sigma(\sigma_{clo}) = \text{clo}(\sigma_e, \sigma_{env'})$ $\quad \Sigma(\sigma_e) = \lambda x.\, \sigma_{body}$

$\Sigma'[\sigma_{env''} \mapsto \sigma_{env'}[x \mapsto \sigma_e]]$

20-3 $\langle \sigma_e, \sigma_{env}, \sigma_k, \Sigma \rangle \overset{MS}{\longmapsto} (\langle \sigma_e', \sigma_{env}', \sigma_k', \Sigma' \rangle)$

$MS(\overset{st=}{\langle \sigma_e, \sigma_{env}, \sigma_k, \Sigma \rangle}) = \langle \sigma_e, \sigma_{env}, \sigma_k, \Sigma' \rangle$

$W = \text{working}(st)$ = mark

$\Sigma' = \text{removeAllBut}(\Sigma, W)$ = sweep

$\text{sweep}(\emptyset, W) = \emptyset$

$\text{sweep}(\Sigma[\sigma \mapsto v], W) = \Sigma''$

where $\Sigma' = \text{sweep}(\Sigma, W)$

$\Sigma'' = \text{if } \sigma \in W$

$\Sigma'[\sigma \mapsto v]$

o.w. Σ'

$\text{mark}(\langle \sigma_e, \sigma_{env}, \sigma_k, \Sigma \rangle) =$

$\text{mark}(\sigma_e, \Sigma) \cup \text{mark}(\sigma_{env}, \Sigma) \cup$

$\text{mark}(\sigma_k, \Sigma)$

$\text{mark}(\sigma, \Sigma) = \text{case } \Sigma(\sigma) \text{ of}$

$b \mapsto \emptyset \mid x \mapsto \emptyset \mid @\, \sigma_L\, \sigma_R \to \text{mark}(L) \cup \text{mark}(R)$

$\lambda x. \sigma_b \mapsto \text{mark}(\sigma_b, \Sigma) \mid \text{clo}(\sigma_e, \sigma_{env}) \mapsto M(\sigma_e, \Sigma) \cup M(\sigma_{env}, \Sigma)$

$\sigma_{env}[x \mapsto \sigma_v] \mapsto M(\sigma_{env}) \cup M(\sigma_v) \mid \text{kret} \mapsto \emptyset$

$\text{kfun}(\sigma_R, \sigma_{env}, \sigma_k) \mapsto M(\sigma_R) \cup M(\sigma_{env}) \cup M(\sigma_k) \mid$

$\text{karg}(\sigma_{clo}, \sigma_k) \mapsto M(\sigma_{clo}) \cup M(\sigma_k)$

n = mem

20-4/ sound = YES

space = fragmentation "optimal"

mark bits $O(\lg n)$

time = malloc ~~free~~ = $O(\lg n)$

mark = $O(\text{work})$

sweep = $O(n)$ (not great)

latency = default bad

but possible to be better (real-time M+S)

21-1/ M&S – time : O(mem) (looks at working+garbage)

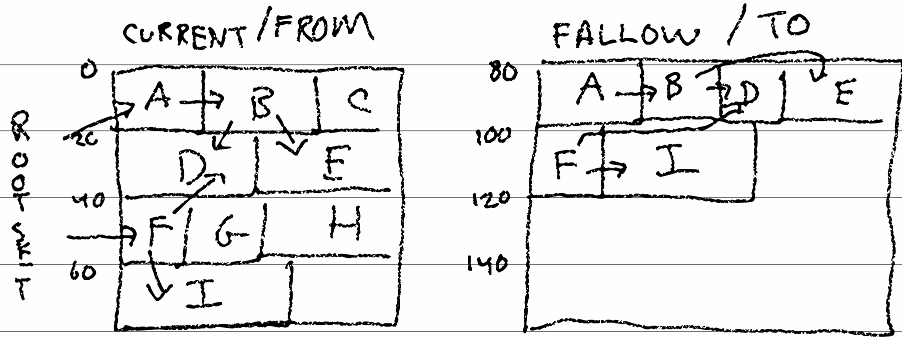
malloc : O(lg n)

space : optimal

Stop + Copy : time : O(working)

malloc : O(1)

space : x2



time 0: rs[x] = A (@0) ⟶ A (@80)

time 1: A.data = B (@5) ⟶ B (@85)

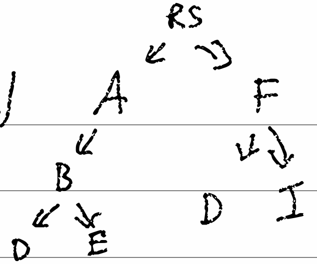
time 3: B.left = D (@20) ⟶ D (@95)

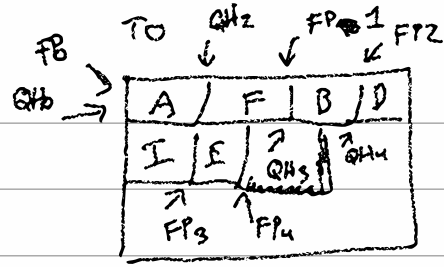
20/D = [OBJ, 42, 50]

95/D = [OBJ, 42, 50]

20/D = [MOVED, 95]

21-2





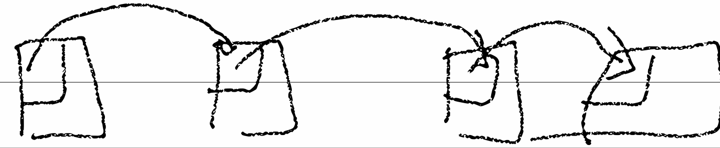
Cheney

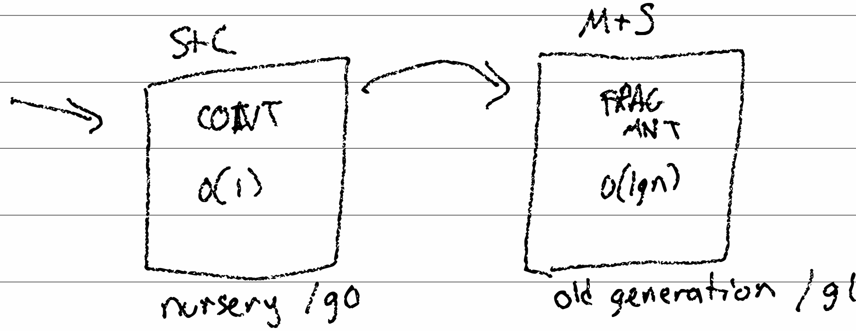
enqueue(o) = store it at FP, update the ref, FWD ptr

dequeue() = read the QH,
move it 1 obj
enqueue children
~~[illegible]~~

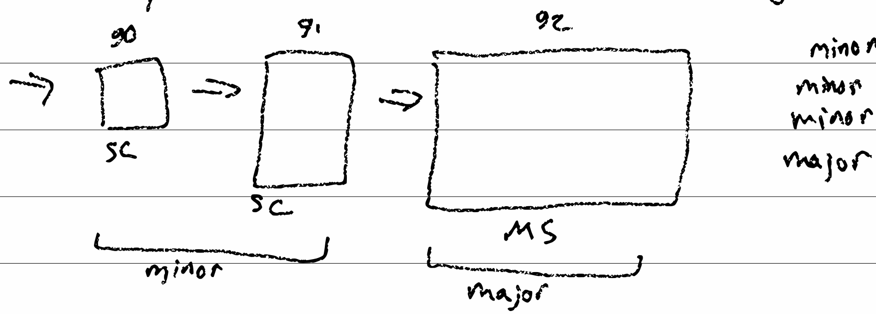
1. enqueue RS
2. while (QH ≠ FP)
   dequeue

22-1)





Generational Hypothesis: Most objects die young.



Inter-generational Pointers: pointers from g0 to g1

or g1 to g0

NEW to OLD - not a problem because major ⇒ minor

OLD to NEW - minor ⇒ ignore g1

cache the references from last major

monitor all ptr creation and add OLD⇒NEW to cache

22-2/ Radioactive Decay Model

Objects have a garbage half-life.

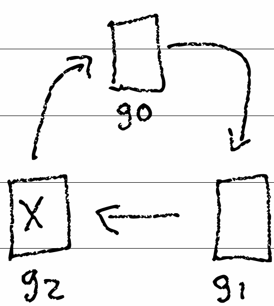
→

The longer you wait, the more garbage you'll find

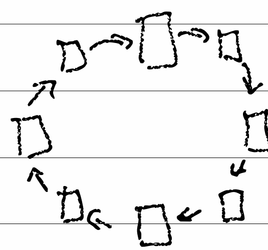
Clinger - Clock Collector

S+C - 2-Clock

3 clock:



8-clock



23-1) errors are undefined behavior

$(+\ 5\ \#\text{true}) \not\Rightarrow$

$(\text{if}\ 5\ 3\ 4) \not\Rightarrow$

$(7\ 3) \not\Rightarrow$

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

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

$E[p\ v \ldots] \rightarrow E[u]$ where $u = \delta(p, v \ldots)$

$\delta(+, 5\ \#\text{true}) = \perp$

(define (+ x y) ((#%+ x y)))

real primitive

(if (and (num? x) (num? y))

. . .

(error ...))

Consequences wrt performance

here: + : any any $\Rightarrow$ num or error

before: + : ~~any~~ ~~any~~ $\Rightarrow$ num or $\perp$

want: + : num num $\Rightarrow$ num

23-2/ A static type system is an analysis of a program that <u>correctly</u> predicts (the values the program evaluates to. | the behavior of the program.)

eval : Program $\Rightarrow$ Ans or $\perp$

type : Program $\Rightarrow$ Prediction

abstract : Ans $\Rightarrow$ Prediction

concrete : Prediction $\Rightarrow$ P(Ans)

abstract 5 = Number concrete bool = {true, false}

abstract true = bool concrete num = {0, 1, -1, ...}

$\forall x.$ concrete(abstract(x)) $\ni$ x

$\forall x. \forall y \in$ concrete(x) $\Rightarrow$ abstract(y) = x

type(pr) = p iff abs(eval(pr)) = p

Soundness theorem

progress: If type(e) = T then e $\to$ e'

preservation: If type(e) = T and e $\to$ e' then type(e') = T

23-3/ $e = num \mid (+\ e\ e) \mid (*\ e\ e)$

$T = Num$

inference rules ① $\forall e.\ e = num \Rightarrow type(e) = Num$

② $\forall e_1, e_2.\ type(e_1) = Num \wedge type(e_2) = Num$

$\Rightarrow type\ (+\ e_1\ e_2) = Num$

③ $\forall e_1, e_2.\ ty(e_1) = Num \wedge ty(e_2) = Num$

$\Rightarrow type\ (*\ e_1\ e_2) = Num$

$\overbrace{(+\ 1\ (*\ 2\ 3))}^{= Num}$ apply rule #2

$ty\ (1) = Num$ $\wedge$ $ty\ (*\ 2\ 3) = Num$

apply #1 apply #3

$ty\ (2) = N$ $\wedge$ $ty\ (3) = Num$

apply #1 apply #1

$e : T$ means $type\ (e) = T$

implicitly $\forall$ vars

$$\frac{a_1 \quad a_2 \quad \cdots \quad a_n}{b_1} := a_1 \wedge a_2 \wedge \ldots a_n \Rightarrow b_1$$

$$n\ \frac{}{num : Num} \qquad a\ \frac{e_1 : Num \quad e_2 : Num}{(+\ e_1\ e_2) : Num}$$

$$m\ \frac{e_1 : Num \quad e_2 : Num}{(*\ e_1\ e_2) : Num}$$

3-24/

$$\frac{\frac{}{1 : \text{Num}}\,n \qquad \frac{\frac{}{2 : \text{Num}}\,n \qquad \frac{}{3 : \text{Num}}\,n}{(*\ 2\ \ 3) : \text{Num}}\,m}{(+\ 1\ \ (*\ 2\ \ 3)) : \text{Num}}\,\text{Add}$$

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

$v = b$ $\qquad\qquad p = \ldots \mid + \mid - \mid * \mid / \mid < \mid = \mid \ldots$

$b = \text{num} \mid \text{bool}$

$T = \text{Num} \mid \text{Bool}$ $\qquad\qquad \Delta(p) = (T_1 \ldots T_n) \Rightarrow T$

$$\frac{}{\text{num} : \text{Num}}\,n \qquad \frac{}{\text{bool} : \text{Bool}}\,b \qquad \frac{e_1 : T_1 \ \ldots\ e_n : T_n}{(p\ e_1 \ldots e_n) : T}\,p$$

$$\frac{e_c : \text{Bool} \quad e_t : T \quad e_f : T}{\text{if}\ e_c\ e_t\ e_f : T}\,i$$

$\Delta(+) = (\text{Num}\ \ \text{Num}) \Rightarrow \text{Num}$

$\Delta(=) = (\text{Num}\ \ \text{Num}) \Rightarrow \text{Bool}$

$\Delta(\text{not}) = (\text{Bool}) \Rightarrow \text{Bool}$

(if (= 5 5) (* -1 (+ 5 0)) (- 5 10))

$$\frac{\Delta(+) = (\text{Num}\ \ \text{Num}) \Rightarrow \text{Num} \qquad \frac{}{5 : \text{Num}}\,n \qquad \text{true} : \text{Num}\ \times}{(+\ \ 5\ \ \text{true})}\,p$$

$$\frac{5 : \text{Bool}\ \times \qquad \frac{}{0 : \text{Num}}\,n \qquad \frac{}{1 : \text{Num}}\,n}{(\text{if}\ \ 5\ \ 0\ \ 1)}\,i$$

23-5/

$$\frac{\Delta(+) = (N\ N) \Rightarrow N \qquad \frac{}{1 : N}\,n \qquad \frac{\frac{\Delta(=) = (N N) \Rightarrow B \quad \frac{}{5:N}\,n \quad \frac{}{5:N}\,n}{(= 5\ 5) : Bool}\,p \qquad \frac{}{6 : N}\,n \qquad \#f : N \ \times}{(if\ (= 5\ 5)\ 6\ \#f) : N}\,;}{(+\ 1\ (if\ (= 5\ 5)\ 6\ false))}\,p$$

$(A \lor B) \Rightarrow p$ iff $A \Rightarrow p$
$\land\ B \Rightarrow p$

$e = v \mid x \mid (p\ e \ldots) \mid (if\ e\ e\ e)$
$\mid let\ x := e\ in\ e$

$v = b$ $\qquad p = \ldots$ $\qquad$ OLD $e : T$

$T = Num \mid Bool$ $\qquad$ NEW $\Gamma \vdash e : T$

$\Gamma : x \Rightarrow T$

$$\frac{T = \Gamma(x)}{\Gamma \vdash x : T} \qquad \frac{\Gamma \vdash e_c : B \quad \Gamma \vdash e_t : T \quad \Gamma \vdash e_f : T}{\Gamma \vdash (if\ e_c\ e_t\ e_f) : T}$$

let $c := (= 5\ 5)$ in
(if c 6 false)

$$\frac{\Gamma \vdash e_x : T_x \qquad \Gamma[x \mapsto T_x] \vdash e_b : T}{\Gamma \vdash let\ x := e_x\ in\ e_b : T}$$

24-1

$$e = \dots \mid (e \ e)$$

$$v = \dots \ \cancel{\mid \lambda x. e} \ \mid \lambda x{:}T.\ e$$

$$T = \dots \mid T \Rightarrow T$$

$$\frac{\Gamma \vdash e_a : T_D \qquad \Gamma \vdash e_f : T_D \Rightarrow T_R}{\Gamma \vdash (e_f \ \ e_a) : T_R}$$

X

$$\frac{\Gamma[x \mapsto T_D] \vdash e_b : T_R}{\Gamma \vdash \lambda x.\, e_b : T_D \Rightarrow T_R}$$

$\forall x \in V,\ e_b \in e,\ T_D \in T,\ T_R \in T,\ \Gamma \in \Gamma s,$
if $\Gamma[x \mapsto T_D] \vdash e_b : \boxed{T_R}$ then
$\Gamma \vdash \lambda x.\, e_b : T_D \to \boxed{T_R}$

$$\frac{\Gamma[x \mapsto T_D] \vdash e_b : T_R}{\Gamma \vdash \lambda x{:}T_D.\, e_b : T_D \Rightarrow T_R}$$

type Γ $(\lambda x.\, e_b) = T_D \Rightarrow T_R$
  where T_R = type $\Gamma[x \mapsto T_D]$ e_b
  T_D = ???

Type taxes

1. Annotations to make type system computable / faster
2. Changing program to make it happy
   (if (= 5 5) 6 false)

24-2 Gradual Typing

$e ::= \ldots \mid (\text{untyped } T\ u)$

$u ::=$ just like e, but no annotations on

$\lambda \mid (\text{typed } e)$

OLD: $\Gamma \vdash e : T$

NEW: $\Gamma \vdash e : T \rightsquigarrow u$

$$\frac{}{\Gamma \vdash num : Num \rightsquigarrow num}$$

$$\frac{\Delta(p) = (T_1 \ldots T_n) \to T \quad \Gamma \vdash e_1 : T_1 \rightsquigarrow e_1' \ \ldots}{\Gamma \vdash (p\ e_1 \ldots e_n) : T \rightsquigarrow (\#\%p\ e_1' \ldots e_n')}$$

$p \to \#\%p$

$+ \to$ unsafe-+

$C = CTC_+(T)$

$$\frac{\Gamma \vdash u \rightsquigarrow u'}{\Gamma \vdash (\text{untyped } T\ u) : T \rightsquigarrow (\text{contract } C\ u'\ \text{"untyped" "typed"})}$$

$\Gamma \vdash u \rightsquigarrow u'$

just goes through u w/ ...

$CTC_+(Bool) = bool?$ $\quad CTC_+(Num) = num?$

$CTC_+(T_D \Rightarrow T_R) = CTC_-(T_D) \Rightarrow CTC_+(T_R)$

$CTC_-(B) = any$ $\quad CTC_-(Num) = any$

$CTC_-(T_D \Rightarrow T_R) = CTC_+(T_D) \Rightarrow CTC_-(T_R)$

$$\frac{\Gamma \vdash e : T \rightsquigarrow u}{\Gamma \vdash (\text{typed } e) \rightsquigarrow u}$$

24-3/ recursion

$$\frac{\Gamma[x \mapsto T_D][rec \mapsto T_D \to T_R] \vdash e : T_R}{\Gamma \vdash (\lambda x. e) : T_D \Rightarrow T_R}$$

data

$$T = \ldots \mid T \times T \mid T + T \mid 1 \mid 0$$

$$\frac{}{\Gamma \vdash tt : 1} \qquad \frac{\Gamma \vdash e_1 : T_1 \quad \Gamma \vdash e_2 : T_2}{\Gamma \vdash \text{pair } e_1\ e_2 : T_1 \times T_2}$$

$$\frac{\Gamma \vdash e : T_1}{\Gamma \vdash \text{inl } \cancel{e} : T_1 * T_2} \qquad \frac{\Gamma \vdash e : T_1 \times T_2}{\Gamma \vdash \text{fst } e : T_1}$$

inl e → inl e T

inr e → inr T e

$$\frac{\Gamma \vdash e : T_2}{\Gamma \vdash \text{inr } T_1\ e : T_1 + T_2} \qquad \frac{\Gamma \vdash e_s : T_1 + T_2 \quad \Gamma[x \mapsto T_1] \vdash e_1 : T \quad \Gamma[y \mapsto T_2] \vdash e_2 : T}{\Gamma \vdash \text{case } e_s \text{ of inl } x \Rightarrow e_1 \text{ inr } y \Rightarrow e_R : T}$$

24-4 $T = \ldots \mid \text{Box } T$

$$\frac{\Gamma \vdash e : T}{\Gamma \vdash \text{box } e : \text{Box } T} \qquad \frac{\Gamma \vdash e : \text{Box } T}{\Gamma \vdash \text{unbox } e : T}$$

$$\frac{\Gamma \vdash e_b : \text{Box } T \qquad \Gamma \vdash e_v : T}{\Gamma \vdash \text{set-box! } e_b \; e_v : 1}$$

$$\frac{\Gamma \vdash e : (\text{Kont } A) \Rightarrow A}{\Gamma \vdash \text{call/cc } e : A}$$

Kont $X =$
$X \Rightarrow 0$
$= X \Rightarrow \text{False}$
$= \neg X$
1 "$=$" True
0 "$=$" False

Curry-Howard Isomorphism

programs	<=>	proofs
:		:
types	<=>	propositions

HOL Isabelle $\boxed{\text{Coq}}$ Agda

24-5/ List $A = 1 + (A \times \text{List } A)$

1: "A" List Num List Bool

2: List A is defined as List A

1 = Polymorphism 2 = Recursive Types

Polymorphism

$T = \ldots \mid A \mid \forall A.T$

$e = \ldots \mid \Lambda A.e$

$\mid e[T]$

$\forall A. A \to A$

$\forall A. A \to \text{Num}$

$\lambda x. x$

"

$\lambda x. 5$

$\Lambda A. \lambda x{:}A. x$

$(e_1 [\text{Num}]\ 6)$

$e_1 = \Lambda A. \lambda x{:}A. 5$

$$\frac{\Gamma, A \vdash e : T}{\Gamma \vdash \Lambda A. e : \forall A. T}$$

$$\frac{\Gamma \vdash T \qquad \Gamma \vdash e : \forall A. T'}{\Gamma \vdash e[T] : T'[A \leftarrow T]}$$

$$\frac{\Gamma \vdash T \qquad \Gamma[x \mapsto T] \vdash e : T'}{\Gamma \vdash \lambda x{:}T. e : T \to T'}$$

Λ

```
class List<X> {
  X elem;
  List<X> next; }
auto l = new List<Num>();
```

$e[T]$

24-6) $T = \ldots \mid \mu A.\, T$

$e = \ldots \mid \text{fold } e \mid \text{unfold } e$

$$\frac{\Gamma \vdash e : T[A \leftarrow \mu A.T]}{\Gamma \vdash \text{fold } e : \mu A.\, T}$$

$$\frac{\Gamma \vdash e : \mu A.T}{\Gamma \vdash \text{unfold } e : T[A \leftarrow \mu A.T]}$$

constructors always call fold

accessors always unfold

"data List A = Empty | Node A (List A)"

$\Rightarrow$ Empty = ... fold ...

Node v l = ... fold ...

caseList l mt node = ... unfold ...

case ... mt

node v l

25-1/ (list (pair "x" 3)
(pair "y" 4)
(pair "name" "yes"))

$\forall X.$ List (Pair Str X))

{x: 3, y: 4; name: "yes")

Pair Num (Pair Num String))

x = fst   y = fst · snd

name = snd · snd

Records

$$e ::= \ldots \mid e.k \mid \langle L_0 = e_0 \ldots L_n = e_n \rangle$$

$$v ::= \ldots \mid \langle L_0 = v_0 \ldots L_n = v_n \rangle$$

$$E ::= \ldots \mid \langle L_0 = v_0, \ldots, L_i = E, L_{i+1} = e_{i+1}, \ldots \rangle$$

$$T ::= \ldots \mid \langle L_0 : T_0, \ldots, L_n : T_n \rangle$$

$$\frac{\Gamma \vdash e : \langle L_0 : T_0, \ldots, L_n : T_n \rangle}{\Gamma \vdash e.L_i : T_i}$$

$$\frac{\Gamma \vdash e_i : T_i}{\Gamma \vdash \langle L_i = e_i \ldots \rangle : \langle L_i : T_i \ldots \rangle}$$

Java: Point

25-2/ (define (dist-from-origin o)

$\sqrt{o.x^2 + o.y^2}$)

(dist-from-origin {x: 3, y: 4})

(d -f -o {x: 3, y: 4, name = "yes"})

"Duck Typing" Structural Typing

vs Nominal Typing

$o : < x : int, y : int >$ — nominal

$o : < x : int, y : int, \underbrace{\ldots}_{\text{allow more}} >$ — structural

Sub-typing = a relation on types

$$T \leq T'$$

$$\{L_0 : T_0, \ldots, L_n : T_n\} \leq \{L'_0 : T'_0, \ldots, L'_m : T'_m\}$$

$$\text{iff } \{(L'_0, T'_0) \ldots (L'_m, T'_m)\} \subseteq \{(L_0, T_0), \ldots, (L_n, T_n)\}$$

$$\frac{\Gamma \vdash e_f : T_D \Rightarrow T_R \quad \Gamma \vdash e_a : T_A \quad T_A \leq T_D}{\Gamma \vdash (e_f \ e_a) : T_R}$$

Liskov Substitution Principle

$T_x \leq T_y$ iff

$\forall e_x \in [T_x] \wedge e_y \in [T_y]$

$\Gamma \vdash C[e_y] : T$

iff $\Gamma \vdash C[e_x] : T$

25-2/ FeedAnimal : Animal $\Rightarrow$ Bool

Feed Cat : Cat $\Rightarrow$ Bool

FeedAnimal Garfield ? ✓

Feed Cat Garfield ? ✓

FeedAnimal Odie ? ✓

Feed Cat Odie ? ✗

Zoo

~~PetHotel~~ : (Animal $\Rightarrow$ Bool) $\times$ Animal $\Rightarrow$ ..

Cat Hotel : (Cat $\Rightarrow$ Bool) $\times$ Cat $\Rightarrow$ ...

			I	T
Zoo	Feed Cat	Garfield ?	✓	✗
CatHotel	FeedAnimal	Odie ?	✗	✗

Feed Cat (Cat $\Rightarrow$ Bool) $\not\leq$ (Animal $\Rightarrow$ Bool)

WRONG

$$\frac{T_{D_1} \leq T_{D_2} \qquad T_{R_1} \leq T_{R_2}}{T_{D_1} \Rightarrow T_{R_1} \leq T_{D_2} \Rightarrow T_{R_2}}$$

RIGHT

$$\frac{T_{D_2} \leq T_{D_1} \qquad T_{R_1} \leq T_{R_2}}{}$$

Foster : $\forall A.$ ($A \Rightarrow$ Bool) $\times$ $A \Rightarrow$ ...

parametricity

25-3/ class BBT <X implements Ordered>

interface Ordered { Bool lt (Ordered, Ordered); }

F-bounded Polymorphism

$T ::= \ldots \mid \top \mid \bot \mid \forall A \leq T.\ T$

$e ::= \ldots \mid \Lambda A \leq T.\ e$

$\bot \leq T \qquad T \leq \top$

$BBT = \Lambda X \leq \text{Ordered}. \ \ldots$

$$\frac{\Gamma \vdash e : \forall A \leq F.\ T' \qquad T \leq F}{\Gamma \vdash e[T] : T'[A \leftarrow T]}$$

25-4/ Type Inference

$f \; x \; := \; x + 5$ untyped

$f \; [x : Int] \; := \; x + 5$ typed w/ annot

$g \; x \; := \; if \; x \; x \; 6$

inference : prog $\Rightarrow$ constraints

constraints $\Rightarrow$ solution

prog $\times$ sol $\Rightarrow$ type

gen : prog $\Rightarrow$ type + constraints

$C = \{T = T_1 \ldots\}$ gen x = x, $\emptyset$

gen n = Num, $\emptyset$

gen $(app \; e_f \; e_a) = T_R, C_1 \cup C_2 \cup \{T_F = T_A \Rightarrow T_R\}$

$T_F, C_1 = gen \; e_f$

$T_A, C_2 = gen \; e_a$

gen $(\lambda x. e) = \cancel{X} \Rightarrow T_R, C_1$

~~gen~~ $T_R, C_1 = gen \; e$

25-5 $\quad f\,x = x + 5$

$\text{gen} \quad \lambda x.\ (+\ x\ 5) = X \to Num,\ ''$

$\text{gen} \quad (+\ x\ 5) = Num,\ \{X = Num,\ Num = Num\}$

$\text{gen} \quad x = X,\ \emptyset$

$\text{gen} \quad 5 = Num,\ \emptyset$

$X \to Num \qquad\qquad \{X = Num,\ Num = Num\}$

$\Rightarrow plug \Rightarrow \qquad\qquad \Rightarrow solve \Rightarrow \quad \{X = Num\}$

$Num \to Num$

Gaussian Elimination $\qquad \{X = Y,\ Y = Z \to A,\ A = Num,\ Z = N \to N\}$

$$\begin{aligned} 5x + 6y &= 10 \\ 3x + 2y &= 3 \end{aligned} \Rightarrow \begin{bmatrix} 5 & 6 & 10 \\ 3 & 2 & 3 \end{bmatrix}$$

$x = -1/4 \quad y = 15/8$

$$\begin{bmatrix} 1 & 0 & -1/4 \\ 0 & 1 & 15/8 \end{bmatrix} \Leftarrow \begin{bmatrix} 1 & 6/5 & 2 \\ 0 & -8/5 & -3 \end{bmatrix}$$

$$2 - \frac{15}{8} \times \frac{6}{5} = 2 - 9/4 = -1/4$$

desugar : Sexpr ~~Ast~~ $\Rightarrow$ AST

26-1/ parsing : String $\Rightarrow$ AST

- yacc bison antlr

BSD Linux Java

" e := NUMBER " NUMBER = [0-9]+

| e PLUS e PLUS = +

| e MULT e " MULT = * "

yacc

parse : String $\Rightarrow$ Sexpr

"5 + 3 * 4" $\Rightarrow$ (5 + (3 * 4))

---

bytecode compile/hl : Program $\Rightarrow$ LL repr

vm/ll : LL repr $\Rightarrow$ ans

LLrepr = C program

= binary format

LLr (Num n) = 0x1 n as 64 bits

LLr (Plus e_1 e_2) = 0x2 (LLr e_1) (LLr e_2)

LLr (Mult e_1 e_2) = 0x3 (LLr e_1) (LLr e_2)

0x2 0x1 0x5 0x3 0x1 0x3 0x1 0x4

~~21533~~ 2 1 5 3 1 3 1 4

262 default : $(\lambda y\ (\lambda x.\ (+\ x\ 5)))$

$\Rightarrow$ LAM "y"\0 LAM "x"\0 ADD VAR "x"\0 NUM 5

distance variables =

LAM LAM ADD VAR 0 NUM 5

---

better low-level in memory

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

Closure

code env

$\lambda x.\ y+z$ $[y \mapsto 6, z \mapsto 8, f \mapsto \ldots, g \mapsto \ldots]$

$\leadsto$ $\leadsto [y \mapsto 6, z \mapsto 8]$

$v0 + v1$ $\leadsto [6, 8]$

AVAR CVAR

26-3/ observational equiv

$X \cong Y$

iff $\forall C.$ $C[X]$ evals a

$\land$ $C[Y]$ evals a

means we can replace X with Y

JIT

vm : prog $\Rightarrow$ ans

compiler : src $\Rightarrow$ prog in asm

hl : src $\Rightarrow$ prog

jit is a version of a vm (prog $\Rightarrow$ ans)

. . . . .

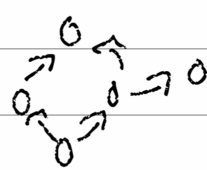
compiler (piece of prog) = asm

run asm

. . . .

---

programs as one thing (theory)



modules + libraries (reality)

modules = records

programs = functions from a record(s) to a main func