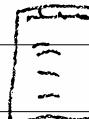


(-1)

# Goal: Implement a VM

what does a program mean?

"1 + 1" means "2"

"word doc" means 

a semantics - english (human)

- math (universal)

- code

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

BNF

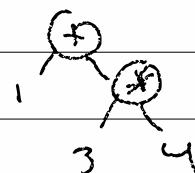
$\mathcal{J}_0$ :  $e := v \mid (+ e e) \mid (* e e)$  or  
 $v := \text{number}$

"(+ 1 (\* 3 4))"  $\in \mathcal{J}_0$ . e

"(+ 1 ))"

$e := v \text{ or } \begin{array}{c} + \\ e \quad e \end{array} \text{ or } \begin{array}{c} * \\ e \quad e \end{array}$

$v := \text{number}$



1-2  = new Add (new Num(1),  
new Num(1))

class E { } new Mult(

class Add : E { } new Num(3),

class Num : E { } new Num(4)) )

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 ~~the~~ language M  
that tells you the semantics of  
language O

Virtual machine : a fast interpreter we like

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

interp v = v

interp ( $+ e_1 e_2$ ) = (interp  $e_1$ )  $+_v$  (interp  $e_2$ )

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

$\leftarrow \text{Num}^* \approx V^*$   
 1-3)  $\Rightarrow$   
 virtual E::interp() = 0;  
 // interp  $V = V$   
 $E^*$  Num::interp() { return this; }  
 $\wedge$  interp ( $+ e_1 e_2$ ) = (interp  $e_1$ ) +  $V$  (interp  $e_2$ )  
 $E^*$  Add::interp() {  
 $\text{Num } n_1 = \text{this.l.interp();}$   
 $\text{Num } n_2 = \text{this.r.interp();}$   
 $\text{return new Num}(n_1 + n_2);$

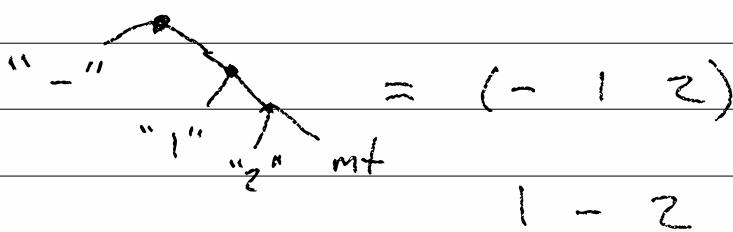
---

$$\begin{aligned}
 (- e_1) &= (* -1 e_1) \\
 (- e_1 e_2) &= (+ e_1 (- e_2))
 \end{aligned}$$

desugar (expander) [compiler] : string tree  
 string expr

$\text{Sexpr} = \text{string or pair of Sexpr}$       S-expr  
 or empty      S-expr

$\rightarrow T_0$



2-1)

Sexpr

$J_0$

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

$$\begin{array}{c} - \\ \diagup \quad \diagdown \\ e_1 \quad e_2 \end{array} \text{mt} \quad (- e_1 \ e_2) = (+ \ (\underline{d} \ e_1) \ (\cancel{*} \ (- e_2))) \\ (+) = 0$$

$$\begin{array}{c} + \\ \diagup \quad \diagdown \\ e_1 \quad \text{more} \end{array} \quad (+ \ e_1 \ . \text{more}) = (+ \ (\underline{d} \ e_1) \ (\underline{d} \ (+ \ . \text{more}))) \\ (+) = 1$$

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

$$J_1 \quad e := v \mid (\text{if } e_1 \ e_2 \ e_3) \mid (e \ e \dots)$$

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

$$\text{prim} := + \mid * \mid / \mid - \mid \leq \mid < \mid = \mid > \mid \geq$$

$$\text{interp } v = v$$

$$\text{interp } (\text{if } e_1 \ e_2 \ e_3) =$$

$$c = \text{interp } e_1$$

$$e_k = \underline{\text{if}} \ c \ \underline{\text{et}} \ \underline{o.v} \ \underline{\text{ef}}$$

$$\text{return}_m \ \text{interp } e_k$$

$$\text{interp } (e_1 \ e_2 \ \dots \ e_n) =$$

$$p = \text{interp } e_1 \ (\text{must be a prim})$$

$$v_0 \dots v_n = \text{interp } v_0 \ \dots \ \text{interp } v_n$$

$$\text{ref } S(p, v_0 \dots v_n)$$

2-2/

$\text{vs delta}(\text{prim } p, \overset{\text{high}}{v^*} \text{ vs}) =$

if ( $p == \text{ADD}$ )

return new Num( $\text{vs}[0].n + \text{vs}[1].n$ )

if ( $p == \text{LT}$ )

return new Bool( $\text{vs}[0].n < \text{vs}[1].n$ )

big-step has a big problem

:  $e \Rightarrow v$

- it is partial

- it says nothing about "in between"

- very un-math-like and clumsy

- inefficient / unhelpful for implementation

small step :  $e \Rightarrow e'$

step (if true  $e_1$   $e_2$ ) =  $e_1$

step (if false  $e_1$   $e_2$ ) =  $e_2$

step ( $P v_0 \dots v_n$ ) =  $S(P, v_0 \dots v_n)$

if step  $e_c = e'_c$  then step (if  $e_c$   $e_1$   $e_2$ )  
= (if  $e'_c$   $e_1$   $e_2$ )

if step  $e_i = e'_i$  then step ( $e_0 \dots e_i e_{i+1} \dots e_n$ )  
= ( $e_0 \dots e'_i e_{i+1} \dots e_n$ )

$$\begin{array}{c}
 \overbrace{(+) + (2+3)}^{2-3} = 2 + \overbrace{(2+3)}^{\text{step}} = 2+5 = 7 \\
 = (1+1) + 5
 \end{array}$$

In context, C = hole

- | if C e e
- | if e C e
- | if e e C
- | (e ... C e ...)

$$\text{plug} : C \times e \rightarrow e$$

$$\text{plug hole } e = e \quad (\text{plug } C e_p)$$

$$\text{plug (if } C e_1 e_2) e_p = (\text{if } e_p e_1 e_2)$$

$$\text{plug } (e_b \dots C e_n \dots) e_p = (e_b \dots (\text{plug } C e_p) e_n \dots)$$

$$\begin{array}{ccc}
 \text{plug } ((+) + (2+3)) & (1+1) & = (1+1) + (2+3) \\
 \tilde{C} & 2 & = 2 + (2+3)
 \end{array}$$

$$\text{Parse} : e \Rightarrow C \times e$$

2.4/

$e \rightarrow e$

step  $C[\text{if true } e \text{ else } e_f] = C[e]$

step  $C[\text{if false } e \text{ else } e_f] = C[e_f]$

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

find-redex :  $e \rightarrow C \uparrow e$

$\uparrow$   
redex

reducible expression

---

When do two programs mean the same thing?

" $1 + 2$ "      " $2 + 1$ "

" $4 \text{ billion} + 1 + 2$ "      " $2 + 1 + 4 \text{ billion}$ "

$\vdash C = \mathbb{B}$

"quicksort"      "mergesort"      "heapsort"

"insertionsort"

$\forall i, \text{"mergesort"} + i = \text{"hs"} i$

introduction      merge       $\vdash C = (\mathbb{B} L)$

$\forall C, C[x] = C[y] \rightarrow \text{observing } (x=y)$   
equivalence

time  $e = \text{days} \times \text{secs}$

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

$C[\text{if true } e_1 \text{ or } e_2] \rightarrow C[e_1]$

$\dots + \dots * \dots - \dots (1+1) \dots * \dots +$

$C := \text{hole} \mid \text{if } C e_1 e_2 \mid \text{if } e_1 C e_2 \mid \text{if } e_1 e_2 C$   
 $(e_1 \dots C e_2 \dots)$

evaluation context, E

$E := \text{hole} \mid \text{if } E e_1 \mid (v \dots E e_2 \dots)$   
E<sub>hole</sub>      E<sub>if</sub>      E<sub>app</sub>

$E[\text{if true } e_1 \text{ or } e_2] \rightarrow E[e_1]$

$E[\text{if false } e_1 \text{ or } e_2] \rightarrow E[e_2]$

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

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

find-redex  $v = \text{false}$

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

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

$(E, e_0) \Rightarrow (\text{if } E e_1 e_2, e_0)$

3-21

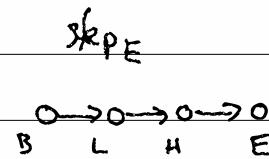
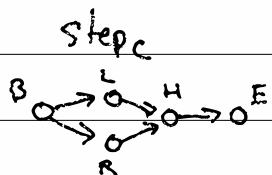
$$fr(v \dots e_0 e_1 \dots e_n) =$$

$$(E, e_0') = f_r e_0$$

$((v \dots E e_1 \dots e_n), e'_0)$

new EApp(v, E, e, .. en)

$$fr(v \dots) = (\text{hole}, (v \dots))$$



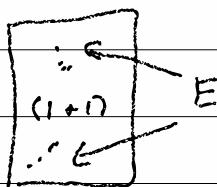
## The Standard Reduction theorem

Alonzo

Curch

(Hue-Rosser)

Rosser

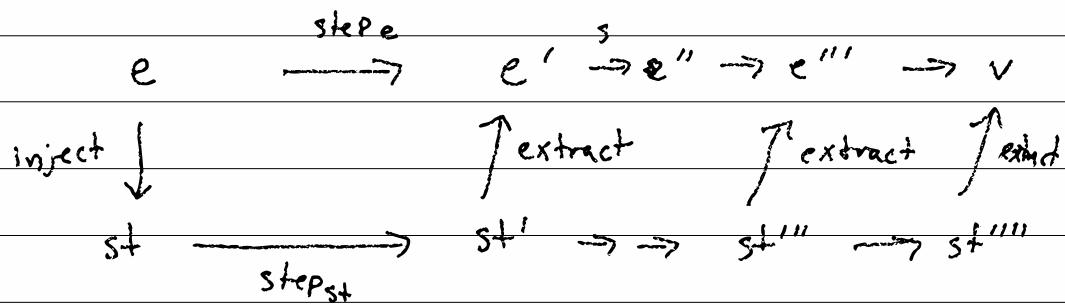


$E = (\text{if } (* \neq 8 (+ 1 2 (* \dots \dots \dots (\text{if } \dots \text{ hole } \dots) \dots)))$

$$\underbrace{E[(1+i)]}_{\text{in time}} \rightarrow \underbrace{E[2]}_{\text{in time}}$$

3-3)  $\text{step}_E = e \rightarrow e$

machine semantics



$$CC_0 \quad st = \langle e, E \rangle$$

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

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

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

- 1  $\langle \text{if } e_c \text{ et } e_f, E \rangle \Rightarrow \langle e_c, E[\text{if hole et } e_f] \rangle$
  - 2  $\langle \text{true}, E[\text{if hole et } e_f] \rangle \Rightarrow \langle e_f, E \rangle$
  - 3  $\langle \text{false}, E[\text{if hole et } e_f] \rangle \Rightarrow \langle e_f, E \rangle$
  - 4  $\langle e_0 e_1 \dots e_n, E \rangle \Rightarrow \langle e_0, E[(\text{hole } e_1 \dots e_n)] \rangle$
  - 5  $\langle v, E[v_0 \dots \text{hole } e_0 e_1 \dots] \rangle \Rightarrow \langle e_0, E[(v_0 \dots v \text{ hole } e_1 \dots)] \rangle$
  - 6  $\langle v_n, E[p \dots \text{hole}] \rangle \Rightarrow \langle \delta(p, v_0 \dots v_n), E \rangle$
- $(y_n, z_n) \rightarrow (l_n, l_n)$

3-4)

HL: ( $\text{test} \ ' (+ 1 (* 2 (\text{if true } 3 4)))$ )  
7)

$\text{test} ( \text{new SExpr} ( \text{new Atom} ("+"),$   
~~(new Se( new A ("1"),~~

....,

),

$\text{new Num} (7))$

$\text{test} ( \text{se}, \text{ex-val}) =$

$e = \text{desugar se}$

$\text{big-step eval } (e) = \text{actual-big-eval}$

$\text{if } (\text{abv} \neq \text{ev}) \{ \text{error} \}$

$\text{small-step eval } (e) = \text{acc-sm-ev}$

$\text{if } (\text{asv} \neq \text{ev}) \{ \text{error} \}$

$\text{cc-eval } e = \text{accv}$

$\text{if } (\text{accv} \neq \text{ev}) \{ \text{error} \}$

$\text{ll-eval } e = \text{allv}$

$\text{if } (\text{allv} \neq \text{ev}) \{ \text{error} \}$

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

$$q-1) \quad C(\_o \rightarrow CK_o$$

$$st = \langle e, k \rangle$$

$$k = k_{ret} \quad // \text{ hole}$$

continuation       $kif\ e\ e\ k \quad // \text{ if } E \times e$

Kontinuation       $kapp(v...) (e...) k \quad // (v... E e...)$

stack k

$$A \rightarrow kapp(v...) (e...) B$$

$$\hookrightarrow B[v \dots A e \dots]$$

$$\text{inject } e = \langle e, k_{ret} \rangle$$

$$\text{extract } \langle e, k \rangle = k_{\text{intoE}}(k)[e]$$

$$\text{done? } \langle v, k_{ret} \rangle = \text{the}$$

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

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

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

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

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

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

$$\delta(SUB, (v_1\ v_0)) \approx v_0 - v_1 \\ (3 \ 4 \ 5)$$

S-1  $J_1 \rightarrow J_2$

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

$x :=$  variable names

$v := b \mid f$

← new

$b :=$  number | bool | prim

$f :=$  function names

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

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

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

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

$J_1:$  step :  $e \rightarrow e$

$J_2:$  step :  $\Delta \times e \xrightarrow{\quad} e$   
 $(f \mapsto d)$

$E[(f \ v \ \dots)] \mapsto E[e[x_0 \leftarrow v] \ \dots [x_n \leftarrow v_n]]$

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

S-2/  $e[x \leftarrow v]$  means look inside of  $e$ ,

find all the  $x$ 's and replace

$$x[x \leftarrow v] = v \quad \text{with } v$$

$$y[x \leftarrow v] = y \quad (y \notin x)$$

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

$$(if \ e_1 \ e_2 \ e_3)[x \leftarrow v] = (if \ e_1[x \leftarrow v] \ e_2[x \leftarrow v] \\ e_3[x \leftarrow v])$$

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

$$\begin{aligned} f(x) &= \overbrace{7x} + \overbrace{2x^2} + 1 \\ f(5) &= 7 \cdot 5 + 2 \cdot 5^2 + 1 \end{aligned}$$

$$\begin{aligned} & (\text{define } (f \ x \ y) \ (+ \ (* \ x \ 2) \ (- \ x \ y))) \Big] = e \\ & (\text{define } (g \ x) \ (f \ x \ x)) \\ & (+ \ 5 \ (g \ 10) \ (f \ 9 \ (g \ 1))) \quad = e \\ & (\ " \ (f \ 10 \ 10) \ " \ ) \\ & (\ " \ (+ \ 10 \cdot 2 \ -10) \ " \ ) \\ & (+ \ 5 \ 10 \ " \ ) \\ & (+ \ 5 \ 10 \ (f \ 9 \ 1)) \\ & (\ " \ (+ \ 9 \cdot 2 \ -1) \ ) \\ & (+ \ 5 \ 10 \ 17) \end{aligned}$$

S-3)  $C_{K_0} \Rightarrow C_{K_1}$  st =  $\langle \Delta, e, k \rangle$

(1+N)

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

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

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

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

(if  $\begin{cases} (x > 10 \text{ mil}) \\ \text{true} \end{cases}$  (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

G-1/ CK<sub>i</sub> is linear-time, so it's not fast !!  
and unrealistic because syntax is available  
at run-time

goal: machine with constant function calls

$\mapsto \langle \Delta, e [x_0 \leftarrow v_0] \dots [x_n \leftarrow v_n], h \rangle$

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

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

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

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

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

## 6-2) CK<sub>i</sub> $\Rightarrow$ CEK<sub>0</sub>

$\text{st} = \langle \Delta, e, \text{env}, k \rangle$

$\text{env} = \text{mt} \quad | \quad \text{env}[x \leftarrow v]$

Jay's env vars c = make-env-empty | make-env-consts( $x, v$ ),  
 $\text{env}$ )

inject<sup>A</sup>e =  $\langle \Delta, e, \text{mt}, k \text{ net} \rangle$

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

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

!!!  $\downarrow$  WRONG  $\downarrow$  !!! (look at head 2 args)

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

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

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

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

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

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

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

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

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

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

$\langle \Delta, v_n, \text{env}, \text{kapp}((P v_0 \dots), (\ ), k) \rangle \mapsto \langle \Delta, \delta(P, (v_0 \dots v_n)), \text{env}, k \rangle$

$\langle \Delta, v_n, \text{env}, \text{kapp}((F v_0 \dots), (\ ), k) \rangle \mapsto \text{let } (\text{define } f \text{ } x_0 \dots) \text{ } e = \Delta \text{ in }$

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

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

(g 10) → (f 5) → (+ 5 y)

$\langle (g 10), \emptyset \rangle \mapsto \langle (f 5), \text{mt}[y \leftarrow 10] \rangle$   
 $\mapsto \langle (+ x y), \text{mt}[y \leftarrow 10][x \leftarrow 5] \rangle$   
 $\mapsto \langle (+ 5 y), \text{ " } \rangle$   
 $\mapsto \langle (+ 5 10), \text{ " } \rangle$   
 $\mapsto \langle 15, \text{ " } \rangle$

WRONG:

$\langle \Delta, (\text{if } e_t \text{ et } e_f), \text{env}; k \rangle$   
 $\mapsto \langle \Delta, e_t, \text{env}; \text{kif}(e_t, e_f, k) \rangle$   
 ~~$\langle \Delta, \text{true}, \text{env}; \text{kif}(e_t, e_f, k) \rangle$~~   
 $\mapsto \langle \Delta, \text{et}, \text{env}; k \rangle$

(define (g y) true)  
(~~def~~ <sup>func</sup> (if (g 10) y 6))  
(f 6)

Dynamic

Scope

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

$\mid \text{kapp}((\lambda \dots), (e \dots), \text{env}, k)$

RIGHT

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

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

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

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

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

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

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

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

$\mapsto \langle \Delta, \delta(p, (v_0 \dots v_n)), \text{mt}, k \rangle$

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

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

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

6-5)  $\mathcal{J}_2 \rightarrow \mathcal{J}_3$

$e := v \mid (e e \dots) \mid (\text{if } e e_1) \mid x$

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

$x := \text{variable names}$

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

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

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

capture-avoiding

$((\lambda(x) x) s) \mapsto s$

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

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

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

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

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

6-6/

$\langle v_n, \text{Dnv}, \text{kapp}((\lambda(x_0 \dots x_n) e) v_0 \dots), (), \text{env} \rangle$   
 $\mapsto \langle e, \text{mt}[x_0 \leftarrow v_0] \dots [x_n \leftarrow v_n], k \rangle$

$\langle ((\lambda x. y. x) s) 6, \emptyset, k \rangle$

$\text{kapp}(((), (6), \emptyset, k))$

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

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

$\langle y, x, (\emptyset[x \leftarrow s]), \text{kapp}(((), (6), \emptyset, k)) \rangle$

$\langle 6, \emptyset, \text{kapp}((y, x), (), \emptyset, k) \rangle$

$\langle x, \emptyset[y \leftarrow 6], k \rangle$

old

$\langle x, (\emptyset[x \leftarrow s])[y \leftarrow 6], k \rangle$

$\langle s, \text{mt}, k \rangle$

$\underbrace{\langle ((\lambda x. y. x) s) 6 \rangle}_{\mapsto} \mapsto (y, 6) 6 \mapsto 5$

CEK

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

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

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

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

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

6-7)

desugar (let ([x<sub>0</sub> e<sub>0</sub>] ... [x<sub>n</sub> e<sub>n</sub>]) be)

= ((λ (x<sub>0</sub> ... x<sub>n</sub>) be) e<sub>0</sub> ... e<sub>n</sub>)

(let ([x s] [y 6]))

(+ x y))

=

((λ (x y) (+ x y))  
s 6)

(let ([x ← s]))

(+ x ↙ ↘ )

(let ↘ ([x (+ 1 x)]) (+ x x))

(+ x y)))

∅[x ← s][x ← 6]

= (+ s

(let ([x ← (+ 1 s)]) (+ x x))

(+ s y))

= (+ s (+ 6 6) (+ s y))

= (+ s 12 q)

= 26

6-8) desugar (let\* () be) =  $\lambda e$   
 desugar (let\* ([ $x_0 e_0$ ] [ $x_1 e_1$ ] ...) be)  
 = (let ([ $x_0 e_0$ ])  
 (let\* ([ $x_1 e_1$ ] ...) be))

(let\* ( $\underbrace{[x \leftarrow s]}$   
 $\underbrace{[y \leftarrow [ (+ x 1) ]}$   
 $\underbrace{[z \leftarrow [ (+ x y) ])}$   
 $(+ z \underbrace{z})$ )

normal let:  $((\lambda (x y z) (+ z z))$   
 $\quad \quad \quad s (+ x 1) (+ x y))$   
 $\text{let* } : \lambda^{(x)} ((\lambda(y) ((\lambda(z) (+ z z) (+ x y)))$   
 $\quad \quad \quad (+ x 1))) s)$

7-1 / ((let ([x←5])  

$$(\lambda(y)(+x^y)))$$
  

$$\overbrace{6)} \Rightarrow (+\ 5\ 6)$$

$$\Delta : e = x \quad | \quad \lambda x. e \quad | \quad e \quad e$$

*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.

## Representas interface

## interface vs Appresentazione

`bool : Opt1 → Opt2 → Some Option 1 or 2`

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

false := Ax. ly. y

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

if tree  $M \circ N \Rightarrow (Ab + f, b + c)$  tree  $M \circ N$

$$\Rightarrow \text{true} \wedge N \Rightarrow (\lambda xy.x) \wedge N \Rightarrow M$$

## 7-2/ Church-encoded numbers

numbers are iteration

num : ThingToDo  $\Rightarrow$  SomethingToDoItTo  
 $\rightarrow$  Result of doing it N times

$$\text{zero} := \lambda f. \lambda z. z$$

$$\text{one} := \lambda f. \lambda z. f z$$

$$\text{two} := \lambda f. \lambda z. f(f z)$$

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

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

$$\Rightarrow \lambda f. \lambda z. f z = \text{one}$$

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

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

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

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

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

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

$$\lambda f. \lambda z. (\text{two } f) (\text{two } f) z$$

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

four

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

$$\begin{aligned} \text{zero? zero} &\Rightarrow \text{zero } (\lambda x. \text{F}) \; \text{T} \\ &\Rightarrow \text{T} \end{aligned}$$

$$\begin{aligned} \text{zero? one} &\Rightarrow \text{one } (\lambda x. \text{F}) \; \text{T} \\ &\Rightarrow (\lambda x. \text{F}) \; \text{T} \\ &\Rightarrow \text{F} \end{aligned}$$

$$\begin{aligned} \text{fst } (\text{pair } M \; N) &\Rightarrow M \\ \text{snd } (\text{pair } M \; N) &\Rightarrow N \end{aligned}$$

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

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

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

$$\begin{aligned} \text{fst } (\text{pair } M \; N) &\Rightarrow (\text{pair } M \; N) \; \text{true} \\ &\Rightarrow (\text{true } M \; N) \Rightarrow M \end{aligned}$$

---

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

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

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

$$\text{inn} := \lambda v. \lambda R. \lambda g. \; g \; v$$

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

copy :=  $\lambda n. n$  add1 zero

74/ sub1 :=  $\lambda n. (n \ F \ (\text{pair zero zero}))$

F :=  $\lambda p. \text{pair} (\text{snd } p) (\text{add1} (\text{snd } p))$

sub :=  $\lambda n. (m, m \ \text{sub1} \ n)$

fac :=  $\lambda n.$

$\lambda$  if (zero? n)

defn

one



copy, not a reference

(mult n (fac (sub1 n)))

mkfac :=  $\lambda \text{fac. }$

$(\lambda n. \text{if} (\text{zero? } n) \text{ one} \text{ ref}$

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

fac :=  $\lambda$  mkfac

$\lambda$  mkfac  $\Rightarrow$  fac

$\lambda (f. m) \Rightarrow m [\text{fac} \leftarrow (\lambda \text{mkfac})]$

$\lambda (x, m) \Rightarrow m (\lambda (x, m))$

$\leftarrow$  fixed-point combinator (eager) aka Y

$Z := \lambda f. (\lambda x. (f (\lambda v. (x \ x \ v))))$   
 $(\lambda x. (f (\lambda v. (x \ x \ v)))))$

$Z \ f = f (Z \ f)$

$$\boxed{Z-5} / \Omega_1 = w \quad w$$

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

$$\text{eg } \Omega \Rightarrow w \quad w \Rightarrow$$

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

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

$$\Rightarrow \quad w \quad w$$

$$\Rightarrow \quad \Omega$$

The Lambda Calculus  $\subseteq S_2$

A  $S_2$  to convert Church to Normal :=

$$\text{Church2Normal} := \lambda n. n \ (\lambda x. (+\ 1\ x)) \ 0$$

$$\text{Church2Normal} \ (\text{fac} \ (\text{succ} \ (\text{plus} \ \text{two} \ \text{two}))) \ 0$$

$$\Rightarrow 120$$

8-1) J<sub>2</sub>: (define (even? x)  
 (if (zero? x) true  
~~(odd? (sub1 x)))~~  
 (define (odd? x)  
 (if (zero? x) false  
 (even? (sub1 x))))  
 (even? 10)

$$\Delta = \{ \text{even?}, \text{odd?} \}$$

✓ odd? is undefined

J<sub>3</sub>: (let\* ([even? (lambda (x) ... odd? ...)])  
 [odd? (lambda (x) ... even? ...)])  
 (even? 10))

J<sub>3</sub>  $\Rightarrow$  J<sub>4</sub> = v := ... | (lambda (x ...) e)

recursive name of fun

(let ([fac (lambda (n) (if (= n 0) 1  
 (\* n (ifac (- n 1)))))])  
 (fac 5))

(desugar (lambda (x ...) e))  $\Rightarrow$  (desugar (lambda (x ...) e))

8-2/ GLD:

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

$$\text{where } \lambda = (\lambda (x_0 \dots x_n) e)$$

NEW:

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

$$\text{where } \lambda = (\lambda f (x_0 \dots x_n) e)$$

CEK<sub>1</sub>  $\rightarrow$  CEK<sub>2</sub>

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

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

$\langle \lambda, \text{env}, k \rangle \mapsto \langle \text{clo}(\lambda, \text{env}), mt, k \rangle$

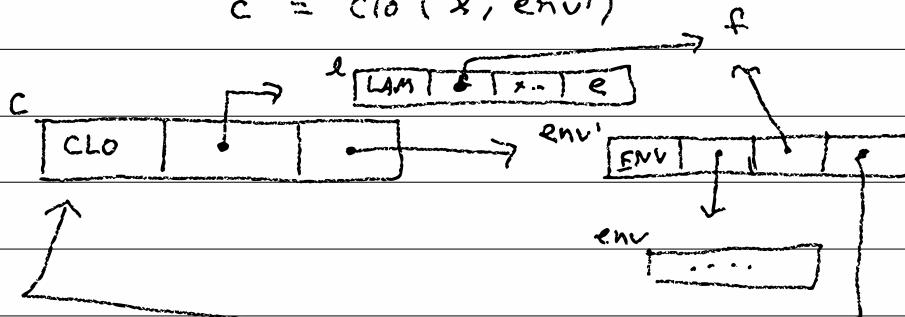
$$\text{where } \lambda = (\lambda (x_0 \dots x_n) e)$$

NEW:  $\langle \lambda, \text{env}, k \rangle \mapsto \langle c, mt, k \rangle$

$$\text{where } \lambda = (\lambda f (x_0 \dots x_n) e)$$

$$\text{env}' = \text{env}[f \leftarrow c]$$

$$c = \text{clo}(\lambda, \text{env}')$$



8-3/  $\text{envp} = \text{make\_env\_ext}(\text{env}, l \mapsto f, \text{NULL});$   
 $\text{clo} = \text{make\_rto}(l, \text{envp});$   
 $\text{envp} \rightarrow \text{val} = \text{clo}; / \backslash \leftarrow \text{installs cycle!}$

as  $\text{nat-unfold} :=$

$(\lambda \text{ rec } (f \in n))$

$(\text{if } (= n 0) z$

$(f n (\text{nat-unfold}^{\text{rec}} f z (-n))))$

$\text{fac } n = \text{nat-unfold } (\lambda (n a) ((\& n a)) 1 n$

## 9-1/ data structures

$v = \dots | p \text{ (primitives)} | b \text{ (constants)}$

Numbers ,  $\neq \in b$  ,  $+ \in P$

Unit :  $\text{ht} \in b$  // Void void  $m \in$

Pairs : pair, fst, snd  $\in P$  (pair  $v_1 v_2 \in V$ )  
 "and"  
 $\wedge v_1, v_2$

(pair 1 2)

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

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

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

Variants : "or" List = empty OR node

$v = \dots | (\text{int } v) | (\text{inn } v)$

unit or (data x lift)

$p = \dots | \text{int} | \text{inn}$

$e = \dots | \text{case } e \text{ as } [(\text{int } x) \Rightarrow e] [(\text{inn } x) \Rightarrow e]$

$E = \dots | \text{case } E \text{ as } [(\text{int } x) \Rightarrow e] [(\text{inn } x) \Rightarrow e]$

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

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

## 9-2/ CEFK ..

$K = \dots | \text{case}_K \text{ env } x_1 e_1 x_2 e_2 K$

$\langle \text{case } e_s \text{ as } [(ml\ x_1) \Rightarrow e_1] [(mr\ x_2) \Rightarrow e_2], \text{env}, K \rangle$

$\mapsto \langle e_s, \text{env}, \text{case}_K \text{ env } x_1 e_1 x_2 e_2 K \rangle$

$\langle \text{inl } v, \dots, \text{case}_K \text{ env } x_1 e_1 x_2 e_2 K \rangle$

$\mapsto \langle e_1, \text{env}[x_1 \leftarrow v], K \rangle$

$\langle \text{inr } v, \dots, \text{case}_K \text{ env } x_1 e_1 x_2 e_2 K \rangle$

$\mapsto \langle e_2, \text{env}[x_2 \leftarrow v], K \rangle$

$\text{Bool} = \text{Unit} \text{ or } \text{Unit}$

$\text{true} = \text{inl } tt$

$\text{false} = \text{inr } tt$

$\text{if } c + f = \text{case } c \text{ as } [(inl -) \Rightarrow f] \\ [(inr -) \Rightarrow f]$

$X \text{ or } Y = \text{Pair Boolean } (X \cup Y)$

$\text{inl } v = \text{pair } \# \text{false} \downarrow v$

$\text{inr } v = \text{pair } \# \text{true} \downarrow v$

$\text{case } e_s \text{ as } [(ml\ x_1) \Rightarrow e_1] [(mr\ x_2) \Rightarrow e_2] =$

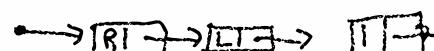
$(\text{let } ([v_s\ e_s]) (\text{if } (\text{fst } v_s) (\text{let } ([x_1\ (\text{snd } v_s)]) e_1) \\ (\text{let } ([x_2\ (\text{snd } v_s)]) e_2)))$

Q-3/ Shape = (circle or (Rect or Triangle))

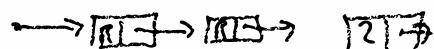
O = int ...



□ = inn (int ...)



△ = inn (inn ...)



## Algebraic Data Types

Type = 1 // Unit ++

O // Nothing

Type × Type // Pair (pair type)

Type + Type // Variant (inl, inr)

B // Base types 5

B = Int32 | ...

Bool = 1 + 1 Nat = 1 + Nat 110

true = int ++ O zero = int ++ two = inn (inr (inr (inr (inr ())))))

false = inn ++ 10 one = inn (int ++)

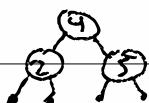
List A = 1 + (A × List A)

mt = int ++ [3, 4] = inn (pair 3

[1] = inn (pair 1 (int ++)) inn (pair 4  
int ++))

Tree A = 1 + (Tree A × A × Tree A)

= inn (pair (inn (pair (pair (int ++) (pair 2 (int ++)))))  
(pair 4 (inn (pair (int ++) (pair 5 (int ++))))) ))



9-4)

leaf = int H

$$\text{node } t_1 \vee t_2 = \text{inn}(\text{pair } t_1 (\text{pair } \vee t_2))$$

single v = node leaf v leaf

 = node (single 2) 4 (single 5)

Option type :       $\text{Maybe } A = \text{None} \mid \text{Just } A$

nothing = int ++

Hash ~~ID~~<sup>ID</sup> (Maybe People) × ID  
→ Maybe (Maybe People)

$\text{seq} : \text{Maybe } A \times (A \rightarrow \text{Maybe } B) \rightarrow \text{Maybe } B$

seq ma f = case ma as

[None  $\Rightarrow$  None]  
[Just a  $\Rightarrow$  f a]

`seq (hash-ref ht 17)`

$(\lambda (x) (\text{hash-ref } h+2 \ x))$

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

empty = init ++

cons a d = inr (pair a d)

1, 2, 3 = cons 1 (cons 2 (cons 3 empty))

map f l =

case l as

[ (init +)  $\Rightarrow$  empty ] ~~(else if)~~

[ (inr x)  $\Rightarrow$  cons (f (fst x)) (map f (snd x)) ]

map add1 (1, 2, 3)  $\Rightarrow$  (2, 3, 4)

filter even? (2, 3, 4)  $\Rightarrow$  (2, 4)

fold f z l =

case l as

[ empty  $\Rightarrow$  z ]

[ cons a d  $\Rightarrow$  f a (fold f z d) ]

sum l = fold + 0 l

sum (1, 2, 3) = (+ 1 (+ 2 (+ 3 0)))

map f l = fold (l (a d) (cons (f a) d)) empty [

## 9-6) Fold fusion

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

=

fold  $(\lambda (a d_2)$

$f_1 \text{ fst } (f_2 a (\text{snd } d_2)) \text{ sn }$

$(z_1 z_2) \text{ l}$

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

sum  $\Sigma$

sum  $\rightarrow$

Haskell

$\text{90-1) } \text{int } x = 7;$   
 $\text{int } y = x;$   
 $x \leftarrow x + 1;$   
 $\text{return } x - y;$

$\left. \begin{array}{l} (\text{let}^* [\Sigma x 7] \\ [y x]) \\ \dots \\ (- x y)) \end{array} \right|$

In C, a variable is a container  
storing a value

In JS, a variable  
is a ~~variable~~ value

A box is a container that might change  
 $p = \dots | \text{box} | \text{unbox} | \text{set-box!}$

$(\text{let}^* ([x b (\text{box } 7)])$   
 $[y (\text{unbox } xb)])$

$(\text{set-box! } xb (+ (\text{unbox } xb) 1))$

$(- (\text{unbox } xb) y)) \Rightarrow 1$

$(\text{set-box! } (\text{box } 7) (+ (\text{unbox } (\text{box } 7)) 1)))$

$(- (\text{unbox } (\text{box } 7)) (\text{unbox } (\text{box } 7)))) \Rightarrow 0$

$(\text{let } ([yb (\text{let } (\text{let } ([xb (\text{box } 0)])$   
 $(\text{set-box! } xb (+ t (\text{unbox } xb))))$   
 $x b)])$   
 $(\text{unbox } yb)) \Rightarrow 7$

$[0-2] / \text{(define } (\text{make-counter})$   
 $\quad (\text{let } ([\text{cb} (\text{box } 0)])$   
 $\quad (\lambda ()$   
 $\quad \quad (\text{set-box! } \text{cb } (+ 1 (\text{unbox cb})))$   
 $\quad \quad (\text{unbox cb}))))$   
 $\quad (\text{let } ([c1 (\text{make-counter})]$   
 $\quad \quad [c2 (\text{make-counter})])$   
 $\quad (\text{list } (c1) (c2) (c2) (c2) (c1)))$   
 $\Rightarrow (\text{list } 1 1 2 3 2 )$

Semantics of  $J_6$

$v = \dots \mid \sigma \quad (\text{pointers})$   
 $\text{old} \Rightarrow : e \rightarrow e \quad E[\cancel{\text{true}} \rightarrow \text{if true } e_1 \text{ else } e_2] \Rightarrow E[e_1]$   
 $\text{new} \Rightarrow : \Sigma \times e \rightarrow \Sigma \times e$   
 $\Sigma : \sigma \rightarrow v$

$\Sigma \times E[\text{if true } e_1 \text{ else } e_2] \Rightarrow \Sigma \times E[e_1]$   
 $\Sigma \times E[\text{box } v] \Rightarrow \Sigma[\sigma \mapsto v] \times E[\sigma] \quad \text{where } \sigma \text{ is fresh}$   
 $\Sigma \times E[\text{unbox } \sigma] \Rightarrow \Sigma \times E[\Sigma[\sigma]] \Rightarrow c$   
 $\Sigma \times E[\text{set-box! } \sigma \text{ } v] \Rightarrow \Sigma[\sigma \mapsto v] \times E[v] \Rightarrow c$

10-3/ Option 2:  $\text{CEK}_3 \rightarrow \text{CESK}$

$\text{CESK} \quad st = \langle e, \text{env}, sto, k \rangle$

$sto = mt \quad | \quad sto[\sigma \mapsto v]$

eg,

$\langle \text{if } ec \text{ et } ef, \text{ env}, sto, k \rangle \mapsto \langle ec, \text{env}, sto, \text{ifk env et effs} \rangle$

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

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

option 2:  $\text{box}$  is a primitive that does

$(\text{box } v) = [V\text{-BOX}, \text{ptr to } v]$

$(\text{unbox } (\text{box } v)) = \text{ret } \underline{\text{ptr}}$

$(\text{set-} \neg \text{box! } [V\text{-BOX}, \text{ptr}], \text{ptr}_z) = \text{changes to memory}$   
 $[V\text{-BOX}, \text{ptr}_z]$

---

$\text{obj\_t* delta\_setbox ( obj\_t* args ) } \in$

$((\text{lobj\_t*}) \text{second(args)}) \rightarrow v = \text{first(args);}$

$\text{return make\_v\_void(); } \}$

(10-4) / Option 1:

$p = \dots | \text{set-fst!} | \text{set-snd!}$

$(\text{let } ([p \ (\text{pair } 1 \ z)])$

$(\text{set-fst! } p \ 3)$

$(\text{fst } p) \Rightarrow 3$

Option 2:

$\text{mpair } xy = \text{pair } (\text{box } x) (\text{box } y)$

$\text{mfst } p = \text{unbox } (\text{fst } p)$

$\text{mset-fst! } p \ nx = \text{set-box! } (\text{fst } p) \ nx$

---

$\text{desugar } (\text{begin}) = (\text{void}) // \text{t+}$

$\text{desugar } (\text{begin } e) = e$

$\text{desugar } (\text{begin } e \ x \ \dots) = (\text{let } ([\_ \ e]) (\text{begin } x \ \dots))$

$\text{desugar } (\text{begin0 } e \ x \ \dots) = (\text{let } ([g \ e]) (\text{begin } x \ \dots \ y))$

$\text{desugar } (\text{when } c \ e \ \dots) = (\text{if } c (\text{begin } e \ \dots) \ \text{void})$

$\text{desugar } (\text{unless } c \ e \ \dots) = (\text{when } (\text{not } c) \ e \ \dots)$

$\text{desugar } (\text{while } c \ e \ \dots) =$

$((\lambda \text{ loop } () (\text{when } c \ e \ \dots (\text{loop}))))$

10-5)

desugar (for  $[x := e_i; e_c; e_f]$   
 $e_b \dots)$

= (let ( $[x \leftarrow e_i]$ ))

(while  $e_c$

$e_b \dots e_f$ ))

(let ([sum (box 0)])

(for [ $x := (\text{box } 0)$ ; ( $\text{unbox } x$ )  $\leq 10$ ; ( $\text{set-box! } x$   
 $+ 1 (\text{unbox } x)$ )]

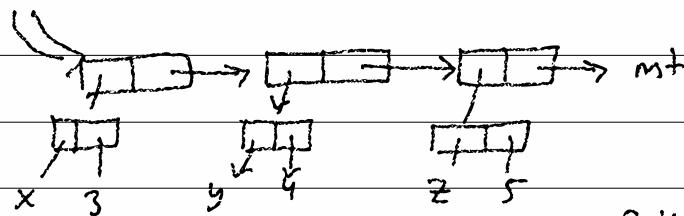
( $\text{set-box! } \text{sum} (+ (\text{ub sum}) (\text{ub } x))$ )

( $\text{unbox sum}$ ))

---

{  $x: 3,$   $\Rightarrow (\text{pair } 3 \text{ } y)$   
 $y: 4 \}$  +  $y$   $x \Rightarrow \text{fst}, y \Rightarrow \text{snd}$

{  $x: 3, y: 4, z: 5 \}$   $\Rightarrow x \Rightarrow \text{fst}, y \Rightarrow \text{fst } \text{snd}, z \Rightarrow \text{snd } \text{snd}$



mtobj) = int tt

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

$\underline{[1-1]}$  (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))  
 $\Rightarrow J_6 \rightarrow J_7 : e = \dots | set! x$   
 $E = \dots | set! x E$

(let ([sum 0])  
 (for [x 0] [< x 10] (set! x (+ 1 x))  
 (set! sum (+ sum x)))  
 sum)  
 (Set! x e)  
 $\Rightarrow$   
 Lvalue = something with an address "o, f = 2" ✓

(set! 2 <) "1~2'x  
 "f(2) = 2" x

11-2

OLD:

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

NEU:

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

$$\emptyset / ((\lambda(x)(+x\neq) 10) \xrightarrow{\text{OLD}} \emptyset / (+ 10 \neq) \\ \xrightarrow{\text{NEU}} \emptyset[\sigma_1 \rightarrow 10] / (+ (\text{unbox } \sigma_1) \neq) \\ \Rightarrow \emptyset[\sigma_1 \rightarrow 10] / (+ 10 \neq)$$

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

(1-3)

The machine implements  $J_6$   
( $\text{set-box!}$ , but no  $\text{set!}$ )

The desugan transforms  $J_7$  into  $J_6$   
( $\text{set!}$ )

desugar  $x = (\text{unbox } x)$

desugar  $(\text{set} \cancel{\text{box!}} x e) \leftarrow (\text{set-box! } x \text{ (desugar } e))$

desugar  $(\lambda (x) e) = \text{if } (\underline{\text{set! } x \_}) \leftarrow e$

$(\lambda (t_0) (\text{let } ([x_0 \text{ (box } t_0)]))$

$(\text{desugar } e))) \quad \underline{\text{o.w.}} \quad (\lambda (x_0) (\text{de } e))$

sugar

$((\lambda (y) (\text{set! } y (+ y 1)) (+ y y)) 10) =$   
 $((\lambda (t_0) (\text{let } ([y \text{ (box } t_0)]))$   
 $(\text{set-box! } y (+ (\text{unbox } y) 1))$   
 $(+ (\text{unbox } y) (\text{unbox } y)))) 10)$

$((\lambda (x) (+ 1 x)) 10) \Rightarrow$

$((\lambda (+) (\text{let } [(x \text{ (box } +)])) (+ 1 (\text{unbox } x)))) 10)$

desugar-top  $e \leftarrow \text{desugar } (\text{mutated-vars } e)$

letrec

11-4) ~~let~~\* ( [ fac (λ rec (n))  
          (if (= n 0) 1  
                  (\* n (fac<sup>rec</sup> (- n 1))))])

[ even? (λ rec (n))

(if (= n 0) true

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

[ odd? (λ ~~rec~~ (n))

(if fn 0) false

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

.... )

$$\begin{aligned}
 \text{11-5) } (\text{let } () \ e) &\stackrel{d}{\Rightarrow} e \\
 (\text{let } ([x_0 \ e_0] \dots [x_n \ e_n]) \\
 e_b) &\stackrel{d}{\Rightarrow} ((\lambda (x_0 \dots x_n) e_b) \\
 &\quad e_0 \dots e_n)
 \end{aligned}$$

$$\begin{aligned}
 (\text{let* } () \ e) &\stackrel{d}{\Rightarrow} e \\
 (\text{let* } ([x_0 \ e_0] \ [x_1 \ e_1] \dots [x_n \ e_n]) \ e_b) \\
 &\stackrel{d}{\Rightarrow} (\text{let } ([x_0 \ e_0])) \\
 &\quad (\text{let* } ([x_1 \ e_1] \dots [x_n \ e_n]) \ e_b)
 \end{aligned}$$

$$\begin{aligned}
 (\text{letrec } ([x_0 \ e_0] \dots [x_n \ e_n]) \ e_b) \\
 &\stackrel{d}{\Rightarrow} (\text{let } ([x_0 \ \text{FALSE}] \dots [x_n \ \text{FALSE}]) \\
 &\quad (\text{set! } x_0 \ e_0) \dots \\
 &\quad (\text{set! } x_n \ e_n) \\
 &\quad e_b)
 \end{aligned}$$

$\underline{11-6}/$  (let ((fac FALSE) [even? FALSE] [odd? FALSE])  
 (set! fac ( $\lambda$  (n) (if (= n 0) 1  
 (\* n (fac (- n 1)))))  
 (set! even? ( $\lambda$  (n) (if (= n 0) TRUE  
 (odd? (- n 1)))))  
 (set! odd? ( $\lambda$  (n) (if (= n 0) FALSE  
 (even? (- n 1)))))  
 (even? (fac 5)))

(letrec ([f ( $\lambda$  (n) (g 0))])

$\Downarrow$  [ $x$  (f 5)]  
 $\Downarrow$  [g ( $\lambda$  (m) m)])

$\times$ )  $\rightarrow$  // fails w/ can't apply boo!

Strategy 1: environment (Racket)

change FALSE to UNDEFINED

Strategy 2: limitry (ML)

restrict RHS of letrec to fun & fn ( $\lambda$ )

Strategy 3: hard to implement (cog / Racket)  
 analyze the program and figure the problem

12-1  $(1 \ 1 \ 0)$  —  $\delta$  is partial and undefined  
on  $\delta(1, (1 \ 0))$

$(5 \ 3)$  — you can't all numbers as has  
(set-box! 7 0) — you can't set-box! numbers

$$\begin{aligned} & (+ (+ 1 1) (+ 2 2)) \\ & \rightarrow (+ 2 (+ 2 2)) \\ & \rightarrow (+ 2 4) \rightarrow 6 \end{aligned}$$

$(5 \ 3) \rightarrow$

$cp = (\text{V-NUM } 3) \quad kp = (\text{K-APP } [(\text{V-NUM } 5)] [ ] \neq \text{K})$

switch ( $cp \rightarrow \text{tag}$ ) // V-NUM

case V-NUM:

switch ( $kp \rightarrow \text{tag}$ ) // K-APP

switch ( $kp \rightarrow \text{vs}[0] \rightarrow \text{tag}$ ) { // V-NUM

case V-PRIM: ...  $\delta(\dots) \dots;$

case V-CLO: ... make new env, lookatcode ...;

default: exit(1);

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

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

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

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

NEU:  $E[u]v \rightarrow \text{"Not a function"} \mid \text{err}$

$u \neq p \text{ and } u \neq (\lambda(x)e)$   $\uparrow$   
an abort

$E[X] \rightarrow E[Y]$  when you throw  
away the context

$\mathcal{S}_7 \rightarrow \mathcal{S}_8: e = \dots \mid \text{abort } e$

$E$  does not contain  $\text{abort } E$

$(+ 1 (+ 2 (\text{abort } (+ 3 (+ 4 0)))))$

$E = (+ 1 (+ 2 \text{ HOLE}))$

$E[(\text{abort } (+ 3 (+ 4 0)))]$

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

CEK:

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

12-3 / int f (int x) {  
 x += 8;  
 return x; } ← k =  
 return is      x \*= 2;      (1)  
 a "local"      x++;      x \*= 2;  
 abort      return x+3;      x++  
 return x+3)  
 3  
 f (5);

(define (fac n)      non-  
 (if (< n 0) (abort "Only  
 (if (= n 0) 1  
 (\* n (fac (- n 1))))) )

12-4 / (+ 1

(try

(+ 2

(throw 3))

$\Rightarrow 10$

catch

$\Rightarrow 6$

(λ (x) (+ x 4)))  $\Rightarrow 8$

$S_g \Rightarrow S_q : e = \dots | \text{throw } e |$

try e catch e

$E = \dots | \text{try } e \text{ catch } E$

| try E catch v

$L = E$  except no try case

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

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

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

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

$e_1 = 3 \quad v_2 = (\lambda (x) (+ x 4))$

$\rightarrow (+ ((\lambda x. (+ x 4)) 3))$

$\Rightarrow (+ (+ 3 4)) \rightarrow (+ 2) \rightarrow 8$

## 12-5) CEK<sub>y</sub> → CEK<sub>5</sub>

kif env e e k	kapp (v..) (e..) env k
K = ...   kTryPre env e k	kret
kTryPost v k	

< try e, catch e<sub>2</sub>, env, fc >

→ < e<sub>2</sub>, env, kTryPre env e, k >

< v, -, kTryPre env e k >

→ < e, env, kTryPost v k >

< v<sub>1</sub>, -, kTryPost v<sub>2</sub> k > → < v<sub>1</sub>, -, k >

< throw e<sub>1</sub>, env, kTryPost v<sub>1</sub> k >

→ < e<sub>1</sub>, env, kAPP (v<sub>1</sub>) () - k >

< throw e<sub>1</sub>, env, kif env' e+lf k >

→ < throw e<sub>1</sub>, env, k >

< throw e<sub>1</sub>, env, kapp (v...) (e...) env' k >

→ < throw e<sub>1</sub>, env, k >

< throw e<sub>1</sub>, env, kTryPre env' e k >

→ < throw e<sub>1</sub>, env, k >

< throw e<sub>1</sub>, env, kret > → < e<sub>1</sub>, env, kret >

12-6)

OLD:  $(5 \quad 3) \rightarrow$

MID:  $(5 \quad 3) \rightarrow \text{err}_e \text{ ("Not a fun")}$

NEW:  $(5 \quad 3) \rightarrow \text{throw err}_e \text{ "Not a fun"}$

$((\lambda (f) \quad (f \ 3)) \ 5)$   
 $(\lambda (f)$   
 $\quad (\text{try } (f \ 3) \ \text{catch } (\lambda (x) \ 15)))$   
 $) \rightarrow 15$

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

(abort)

$\text{J}_q \ni \text{try}, \text{throw}$

13-1]  $\text{J}_8 \rightarrow \text{J}_{10}$

cf

$e = \dots / \text{call/cc } e$

$$E[\text{call/cc } v] \Rightarrow E[v (\lambda(x) (\text{abort } E[x]))]$$

(+ 1

$(\text{call/cc } (\lambda(\text{esc}))$

$(\text{let } ([\text{throw } (\lambda(y) (\text{esc } (+ y 4)))])$   
 $(+ 2 (\text{throw } 3))))))$

$$\Rightarrow E = (+ 1 \bullet) \quad "E[(\text{call/cc } \text{J})]$$

$(+ 1 ((\lambda(\text{esc}) \dots) (\lambda(x) (\text{abort } (+ 1 x)))))$

$\Rightarrow (+ 1 (+ 2 ((\lambda(y) (\lambda(x) (\text{abort } (+ 1 x)) (+ y 4)))$   
 $3))))$

$\Rightarrow (+ 1 (+ 2 ((\lambda(x) (\text{abort } (+ 1 x))) 7)))$

$\Rightarrow (+ 1 (+ 2 (\text{abort } 8 + 1 7)))$

$\Rightarrow (+ 1 7) \Rightarrow 8 \quad E[\text{abort } e] \mapsto e$

[32] desugar (try e<sub>1</sub> catch e<sub>2</sub>)  $\Rightarrow$

try-catch ( $\lambda ()$  e<sub>1</sub>) e<sub>2</sub>

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

standard library:

throw := ( $\lambda (x)$  ((unbox last-handler) x))

last-handler := (box ( $\lambda (x)$  (abort x)))

try-catch := ( $\lambda$  (body new-handler))

(let ([old-handler (unbox last-handler)]))

(begin0 (let/cc here (set-box! last-handler

( $\lambda (x)$  (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))))))
```

$\Rightarrow$

```
desugar (define (f x ...) b)
          $\Rightarrow$  (define (f x ...) (let/cc return b))
```

### break/continue:

OLD

```
desugar (while c b)  $\Rightarrow$  ((λ loop () (when c b (loop))))
```

NEW  $\Rightarrow$  ((λ loop ()

(when c

(let/cc break (let/cc continue b)

(loop))))))

[3-Y] R<sup>5</sup>RS (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."

$$\text{CEK}_y \xrightarrow{\text{(about)}} \text{CEK}_z \quad (\text{call/cc})$$
$$v = \dots \mid \text{kont } k$$

$$< v, \dots, \text{kApp} [\text{call/cc}] [] - k >$$

$$\mapsto < \text{kont } k, \dots, \text{kApp} [v] [] - k >$$

$$< v, \dots, \text{kApp} [\text{kont } k] [] - - >$$

$$\mapsto < v, \emptyset, k >$$

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

CPS - continuation-passing style

input:

$(+ 1 \ (\text{call/cc} \ (\lambda \ (\text{esc}) \ (+ 2 \ (\text{esc} \ 3))))))$

old:  $+ : V \times V \rightarrow V$       call/cc:  ~~$(V \rightarrow A) \times (V \rightarrow A)$~~   
 $(\lambda \rightarrow V) \rightarrow V$

New:  $+ : V \times V \times (V \rightarrow A) \rightarrow A$

call/cc:  $((V \rightarrow A) \rightarrow A) \times (V \rightarrow A) \rightarrow A$

call/cc :=  $(\lambda \ (f \ k))$

$(f \ (\lambda \ (v \ n k) \ (k \ v)) \ k))$

$(\text{call/cc} \ (\lambda \ (\text{esc} \ k))$

$(\text{esc} \ 3 \ (\lambda \ (\text{ans}) \ (+ 2 \ \text{ans} \ k))))$

$(\lambda \ (\text{ans}) \ (+ 1 \ \text{ans} \ \text{top})))$

$\Rightarrow (\text{esc} \ 3 \ (\lambda \ (\text{ans}) \ (+ 2 \ \text{ans} \ k))))$

$[\text{esc} \mapsto (\lambda \ (v \ n k) \ (+ 1 \ v \ \text{top})),$

$k \mapsto (\lambda \ (\text{ans}) \ (+ 1 \ \text{ans} \ \text{top}))]$

$\Rightarrow (+ 1 \ 3 \ \text{top}) \Rightarrow (\text{top} \ 4) \Rightarrow 4$

(3-6) CPS-m  $\Rightarrow$  ~~give the answer~~

$$\alpha = \lambda | x$$

$$\text{call} = (\alpha \dots)$$

$$v = b | (\lambda (x \dots) \text{call})$$

$$\Theta = (\lambda (\text{top}) \text{call})$$

$$<(\lambda (x \dots) \text{call}), \text{env} v>$$

$\mapsto <\text{clo}($

It's simple!

$$\alpha = v | x$$

$$\text{call} = (\alpha \dots)$$

$$v = b | (\lambda (x \dots) \text{call})$$

$$st = <\text{call}, \text{env}>$$

$$<(\alpha_0 \alpha_1 \dots \alpha_n), \text{env}>$$

$$\mapsto <c, \text{env}'>$$

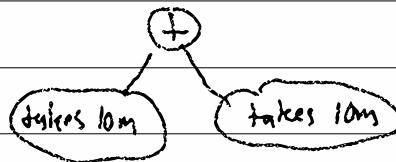
where  ~~$\alpha$~~   $\alpha(\alpha_0) = (\lambda (x_0 \dots x_n) c)$

$$v_1 \dots v_n = \text{map } \alpha (\alpha_1 \dots \alpha_n)$$

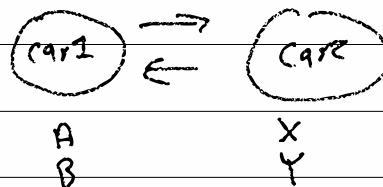
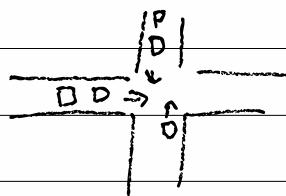
$$\text{env}' = \emptyset [x_1 \mapsto v_1] \dots [x_n \mapsto v_n]$$

$$\alpha v = v \quad \alpha x = \text{env}[x]$$

(S-1)) Concurrency - logical simultaneity  
Parallelism - concrete simultaneity



Serial / Sequential - 20m  
Parallel - 10m



possible programs: A B X Y - P → Z  
X Y A B - Z → I

Concurrency has N ans

A X B Y - interleaving 2 Ans  
= 2<sup>st</sup>

Parallel has 1 ans

X A Y B  
X A B Y - car1 / car2

A X Y B - car1 / car2

A Y X B X

15-2 spawn! : ( $\rightarrow a$ )  $\rightarrow$  void

exit! : ()  $\rightarrow$  void

$\leftarrow$  = happens before

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

(spawn! (λ () (print 'X)  
                  (print 'Y)))  
(exit!))

(define RQ empty) ;; RQ = ready queue

(define (spawn! +)

(set! RQ (cons (λ () (begin (+) (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  $\xrightarrow{x}$  20  $\xrightarrow{+}$  22

10  $\xrightarrow{+}$  12  $\xrightarrow{x}$  24

make-channel : () → Chan

send! : Chan × A → ()

recv! : Chan → A

(let ([ch (make-channel)])  $\quad \downarrow A_1$   $\quad \downarrow A_2$  (send! ch B))

(spawn! (λ () (send! ch 20)))  $\quad \downarrow B$

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

(exit!))

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

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

$ABC^{=22}$   $BAC^{=18}$   $CAB^{=22}$

$ACB^{=18}$   $BCA^0$   $CBA^D$

~~A~~ 22, 8 — B, C

18, 20 — C, B

15-4 make-chan :  $() \rightarrow \text{chan } A$   
 send! :  $\text{chan } A \times A \rightarrow ()$   
 recv! :  $\text{chan } A \rightarrow A$

$\text{Chan } A = \text{Box} \left( \begin{array}{l} \text{Empty}(\text{channel}) \\ \leftarrow \\ 1 + (A \times (0 \rightarrow B) \times \text{Chan } B) \\ + ((A \rightarrow C) \times \text{Chan } A) \end{array} \right)$

$\nwarrow$  sendch  
 $\nearrow$   
 $\nearrow$   
 $\nearrow$  Rcv ch

make-chan := (box  $\lambda$ )

send! ch v :=

case (unbox ch) of

RecvCh f ch'  $\rightarrow$  (spawn! (λ () (f v)))  
 (set-box! ch ch')

next  $\rightarrow$  (let/cc me

(set-box! ch (SendCh v me next))  
 (exit!))

Recv! ch :=

case (unbox ch) of

SendCh v f ch'  $\rightarrow$  (spawn! f)

(set-box! ch ch') ; v

next  $\rightarrow$  (let/cc me

(set-box! ch (RecvCh me next))  
 (exit!))

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

make-promise<sup>f</sup> := (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 bool's")

#1 → 2 ((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+ = #%+

(+ box? "Not a number")))

16-2/

(define (f x) (\* x 2))

(f 1 2)

desugar (e<sub>0</sub> ... e<sub>n</sub>) = (SAFE-(ALL e<sub>0</sub>  
(list e<sub>1</sub> ... e<sub>n</sub>))

→ (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"))])

v<sub>2</sub>

<del>, -, kApp ([#%apply ~~del~~] []) - k >

→ <del vN, -, kApp [v<sub>0</sub> v<sub>1</sub> ... v<sub>n-1</sub>] [] - k >

where v<sub>2</sub> = (list v<sub>1</sub> ... v<sub>n</sub>)

(apply + (list 2 3)) ⇒ (+ 2 3) ⇒ 5

16-3) bst-insert : BST? × Num?  $\rightarrow$  BST

(define (bst-insert b v)

(unless (BST? b) (throw ...))

(unless (num? v) (throw ...))

... )

MeD :

unsafe-f<sub>1</sub> : ... unsafe-f<sub>2</sub> ...

f<sub>2</sub>

f<sub>3</sub>

f<sub>1</sub> : check(unsafe-f<sub>1</sub>, ...)

f<sub>2</sub> : check (unsafe-f<sub>2</sub>, ...)

protect

+

-

apply-Contract : Value × Contract × Blame × Blame

(define bst-insert

(apply-contract unsafe-bst-insert

(fun-cte (list BST? num?))

BST?)

"Me" "Them" ))

( with-module

16-4) desugar (module  
body ...)

# exports

[f cte] ...) user ...)

$\Rightarrow$  (apply ( $\lambda$  (f ...) user ...)

(letrec ( body ...))

(list (contract f cte ME THEM)  
... )))

Contract = FlatCtc ( $A \rightarrow \text{Bool}$ )

FunCtc (List contract) contract

BSR? = FlatCtc ( $\lambda (x) \dots$ )

protect  $\vee$  (FlatCtc pred) pos neg =

(if (pred  $\vee$ )  $\vee$  (error "pos is attack"))

protect  $\vee$  (FunCtc dom rng) pos neg =

(if (! (function? v) (= (length dom) (fun-arity A)))

(error pos "not a fun or wrong arity"))

( $\lambda$  args

(if (! (= (len args) (length)) (error neg)))

(protect (apply  $\vee$  (map ( $\lambda (a cte)$  (protect a cte

rng pos neg)))))))

len dom))

neg pos))

16-5/

(product map)

$\oplus$  (FunCtc  $\ominus$  (list  $\oplus$  (FunCtc  $\oplus$  (list Num?))  
 $\ominus$  Bool?))

$\oplus$  (ListCtc  $\bullet$  Num?)  
 $\oplus$  (ListCtc Bool?)))

## 17-V Syntactic Extension

(time e)  $\Rightarrow$  return value of e  
and print how long it took  
to run

now  $\in P$       (define (time' x)  
                  (letx ([before (now)])  
                  [ans (x)])  
E [(lx. b) v]  $\Rightarrow$  [ans (x)]  
E [b[x  $\leftarrow$  v]]  $\Rightarrow$  [aftr (now)])  
                  (displayln (- aftr before))

(time (fib 100))

ans))

(time' () () (fib 100)))

§

cpp

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

(macros operate on text of programs  
NOT values of the program)

## 17-2/ Cpp problems

#define SUB -2

...

return x SUB;  $\Rightarrow$  x - 2

...

#define FOO 2);

...

printf("The number is %d", FOO  
return 1;

"(+ 2 2)"  $\Leftarrow$



text editor

program tools (compiler)

17-3/ macros : syntax  $\rightarrow$  syntax (like desugaring)  
macros - by example (1986) - we'll implement

(define-syntax-rules let  $\leftarrow$  macro name  
[ (let ([x xe] ...) be ...)  $\leftarrow$  pattern  
((lambda (x ...) be ...) xe ...)])  $\leftarrow$  template

(dsr name [pat template] ...)

expand : Set of Rules  $\times$  SExpr  $\rightarrow$  SExpr

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

expand  $\Sigma$  (cons mac rest) =

if  $\Sigma$  (mac) = defn then

~~expand~~ expand defn (cons mac rest)

else

App ( expand  $\Sigma$  mac , map (expand  $\Sigma$ ) rest)

expand  $\Sigma$  (Num n) = (Num n)

expand  $\Sigma$  (cons 'lambda (cons args body)) =  
Lambda args (expand  $\Sigma$  body)

[74] expand I

pato

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

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

match pato use =

[x ↦ (a b)]

[xe ↦ (3 4)]

[be ↦ ((+ a b) )]

match : Pat × Sexpn ⇒

Env or #F

transcribe tempo  $\overset{?}{=}$

((λ (a b) (+ a b)) 3 4)

expand  $\sum$  (define-syntax-rules mac defns)  
 $= \sum [\text{mac}] = \text{defns}$

17-5/

match : Pat  $\times$  Sexpr  $\rightarrow$  Env or false

match () () =  $\emptyset$

match b b =  $\emptyset$

match x se =  $[x \mapsto (\emptyset, se)]$

match (p ...)  $\not\models$  se =

merge (map (match p) se)

match (cons p1 pr) (cons l r) =

match p1 l  $\cup$  match pr r

merge : List (Env or false)  $\rightarrow$  Env

merge [] =  $\emptyset$

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

merge [ $\sigma_0 \ \sigma_1 \ \dots$ ] =

new hash where  $k \dots =$  keys of  $\sigma_0$

$k \mapsto (\text{level}+1, vs)$

where  $\sigma_0(k) = (\text{level}, -)$

$vs = (\text{init } \text{sh}(\sigma_0(k)) \ \text{and } (\sigma_1(k)), \dots)$

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~~ b))]

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

(77) transcribe : env  $\times$  template  $\Rightarrow$  sexpr

transcribe  $\sigma$  () = ()

transcribe  $\sigma$  b = b

transcribee  $\sigma$  x = se where  $\sigma(x) = (0, se)$

tr  $\sigma$  (p ...) =

map ~~lambda~~ ( $\lambda (\sigma')$  (tr  $\sigma'$  p))  
(decompose  $\sigma$ )

tr  $\sigma$  (cons t<sub>1</sub> t<sub>2</sub>) = (cons (tr  $\sigma$  t<sub>1</sub>)  
(tr  $\sigma$  t<sub>2</sub>))

decompose : env  $\Rightarrow$  list env

decompose  $\sigma$  = (list  $\sigma_0 \dots \sigma_n$ )

where  $n = \text{length of } \sigma[x]$  for some x

$\sigma_i[x] = (\text{level} - i, \text{vs}[i])$

where (level, vs) =  $\sigma[x]$

[7-8] (define -syntax-rules or  
[or true]  
[or x y ...)  
(let ([tmp x])  
(if tmp true (or y ...))))])

(or false  $\stackrel{?}{3}$  y)  $\Rightarrow$  3  
(begin (display "H;!")  
3)

(let ([tmp<sub>rd</sub> 7])  
(or false tmp<sub>rd</sub>))  $\Rightarrow$  7  
(let ([tmp<sub>rd</sub> 7])  
(let ([tmp<sub>bi</sub> false])  
(if tmp<sub>bi</sub> tmp<sub>bi</sub>  
(let ([tmp<sub>gr</sub> tmp<sub>rd</sub>])  
(if tmp<sub>gr</sub> tmp<sub>gr</sub> false))))))  $\Rightarrow$

macro : sdt  $\rightarrow$  std

# 18-1/ Backtracking Non-determinism

$\text{nde} = (\text{ans } v)$	$= \text{Ans}$
fail	$= \perp$
(choice $\text{nde}$ $\text{nde}$ )	$: (\text{NDA}) \times \text{NDA} \rightarrow \text{NDA}$
(bind $\text{nde}$ $f$ )	$: \text{NDA} \rightarrow (\text{A} \rightarrow \text{NDB}) \rightarrow \text{NDB}$

$\text{run} : \text{ND A} \rightarrow \cancel{\text{list}} \text{ stream A}$

$\text{empty-stream} = \text{empty}$

$\text{stream-cons} : a \times (\rightarrow \text{stream a}) \rightarrow \text{stream a}$

$\text{stream-fst} : \text{stream a} \rightarrow a$

$\text{stream-rest} : \text{stream a} \rightarrow \text{stream a}$

$\text{run p} = \text{sols} (\text{list} (\text{st p} (\text{kont:return})))$

$\text{sols '()} = \text{empty-stream}$

$\text{sols} (\text{cons} (\text{st p k}) g) =$

$\text{case p of (bind p' f)} \Rightarrow \text{sols} (\text{cons} (\text{st p'} (\text{kont:bind f k})) g)$

$(\text{choice p}_1 \text{ p}_2) \Rightarrow \text{sols} (\text{cons} (\text{st p}_1 \text{ k}) (\text{st p}_2 \text{ k}) g)$

$(\text{fail}) \Rightarrow \text{sols } g$

$\text{ans v} \Rightarrow \text{case k of return} \Rightarrow \text{stream-cons v}$

$\text{bind f k} \Rightarrow \text{sols} (\text{cons} (\text{st} (\text{f v}) \text{ sc}) g)$

18-2 query : Question  $\rightarrow$  ND ans

query  $g = \text{bind} (\text{searchN DB } \emptyset (\text{fst } g))$   
 $(\lambda (\text{env}) (\text{ans} (\text{transcribe env} g)))$

SearchV : Rules  $\times$  Env  $\times$  List(Questions)  $\rightarrow$  ND Env

SearchN rules env  $\boxed{g}$  = ans env

$g : g_3 = \text{bind} (\text{searchN rules env } g_5)$   
 $(\lambda (\text{env}') (\text{Search* rules env' rule}_6))$

Search\* : Rules  $\times$  Env  $\times$  Rules  $\times$  Question  $\rightarrow$  ND Env

Search\* all env some  $g = \text{case some of}$

$\boxed{\cdot} \Rightarrow \text{fail} \quad | \quad \text{rule}_N : \text{rules} \Rightarrow$   
choice (search\* all env rules  $g$ )  
(search1 all env rule $_N$   $g$ )

Search1 : Rules  $\times$  Env  $\times$  Rule  $\times$  Question  $\rightarrow$  ND Env

Search1 all env (conclusion, deps)  $g =$

(bind (match conclusion  $g$ )  
 $(\lambda (\text{env}') (\text{searchN all env env' deps}))$ )

## 18-3/ Non-determinism Expressions are a Monad

Monad A =

return :  $A \rightarrow \text{Monad } A$

bind :  $\text{Monad } A \rightarrow (A \rightarrow \text{Monad } B)$   
 $\Rightarrow \text{Monad } B$

List Monad

return  $x = [x]$

bind  $mx f = \text{map } f mx$

bind (return 1) add1  $\Rightarrow [2]$

bind (list 1 2 3) add1  $\Rightarrow [2 3 \rightsquigarrow]$

bind x (λ(a))

bind ~~(list~~ (add1 a) (sub1 a))  
mult ))  $\Rightarrow$

# 19-1 / Memory Management

$$e = \text{lit} \mid (e \ e) \mid x$$

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

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

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

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

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

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

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

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

soundness  $\leftarrow$  right wrong 00m cash

efficiency  $\leftarrow$  { space  
    time }

performance

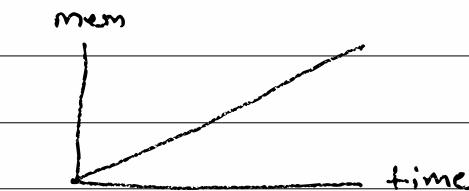
[9-2] soundness - **right** writing **6cm** **graph**  
efficiency

- Space

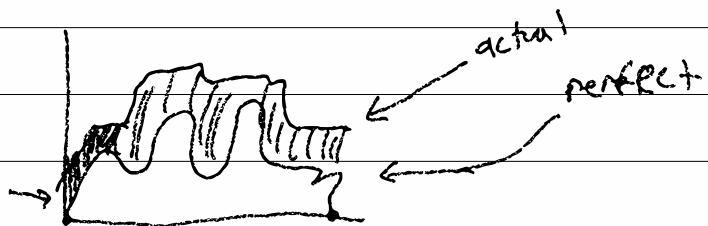
mem

infinite mem

peak usage = highest point



differ  
space eff



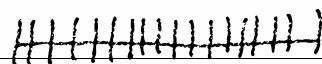
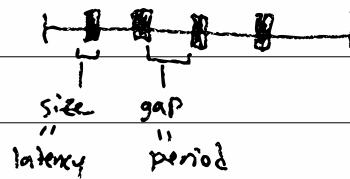
- time

start  $\xrightarrow{\text{end}}$

perfect

$\xrightarrow{\text{---}}$  more time for MM

performance

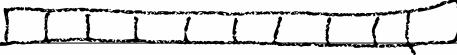


low latency



high latency

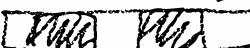
## 14-3/ C-style malloc & free

mem = 

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

free (ptr) = figure out adj free space

$O(\lg n)$

 = Fragmentation

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

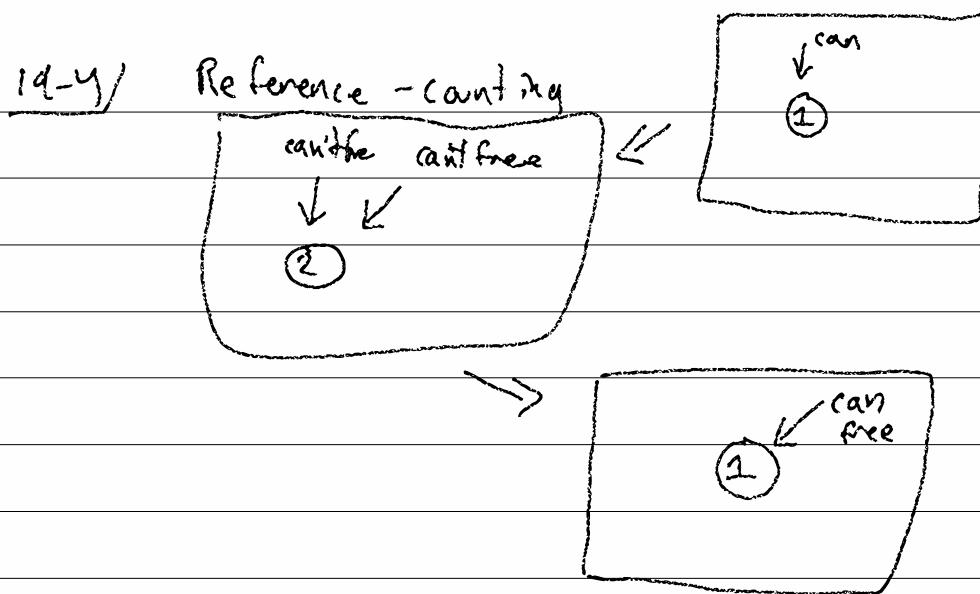
space eff =  $O(\lg n)$

safeness = not depend on programs

performance = •

safely:

turn all aliases into copies (half copy, XCopy)



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

sound: who calls ref/unref? if programmer  $\Rightarrow$  no sound  
space eff: every object stores count if lang  $\Rightarrow$  sound

$$LL(16) \Rightarrow (16 + CNT) \quad CNT = [1, 8]$$



~~cycles are never deleted~~

time eff:  ~~$\propto$~~  & work of program  $O(w)$  not  $O(n)$

latency : cascading deletes of large trees = high

Name: \_\_\_\_\_ w/ grades

Lafency

# 20-1) Garbage Collection (a sound memory manager)

John McCarthy for LISP in 1959

## Mark & Sweep

...

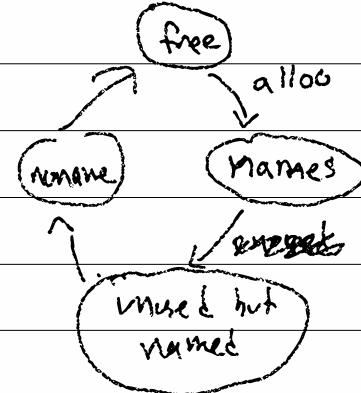
var x = ...;

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

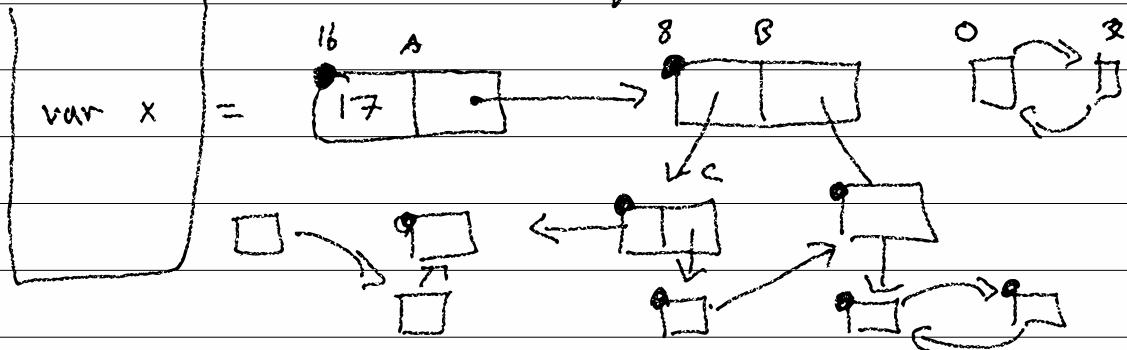
line 21: if (x) {  
  ... x ...;

[ 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

20-2/ mark = from root set, mark all children recursively  
 sweep = go through mem, delete if no mark  
 remove marks o.w.

$$\begin{aligned}
 \text{CESK+MS} : \quad & st = \langle \sigma, \sigma, \sigma, \Sigma \rangle \quad \sigma = \emptyset \mid n \\
 & \Sigma = \cancel{\Sigma} \Sigma [\sigma \mapsto *] \quad k = \text{kref} \\
 & e = v \mid @ \underset{\sigma}{\sigma} \sigma \mid x \mid \text{kfun}(\sigma, \sigma) \\
 & v = b \mid \boxed{\lambda x. \sigma} \mid \text{clo}(\sigma; \sigma) \mid \text{karg}(\sigma, \sigma) \\
 & \text{env} = \cancel{\Sigma} \sigma [x \mapsto \sigma]
 \end{aligned}$$

$$\begin{aligned}
 < \sigma_e, \sigma_{\text{env}}, \sigma_k, \Sigma > = \text{case } \Sigma(\sigma_e) \text{ of} \\
 & x \rightarrow < \Sigma(\sigma_{\text{env}})[x], \cancel{\emptyset}, \sigma_k, \Sigma > \\
 & \lambda x. \sigma_b \rightarrow < \sigma_{\text{clo}}, \emptyset, \sigma_k, \Sigma' > \quad \Sigma' = \Sigma \left[ \sigma_{\text{clo}} \mapsto \text{clo}(\sigma_e, \sigma_{\text{env}}) \right] \\
 & @ \sigma_l \sigma_r \rightarrow < \sigma_l, \sigma_{\text{env}}, \sigma_{k'}, \Sigma' > \quad \Sigma' = \Sigma \left[ \sigma_{k'} \mapsto \text{kfun}(\sigma_r, \sigma_{\text{env}}, \sigma) \right] \\
 & \_ \rightarrow \text{case } \Sigma(\sigma_k) \text{ of}
 \end{aligned}$$

$$\begin{aligned}
 & \text{karg}(\sigma_{\text{clo}}, \sigma_{k'}) \rightarrow < \sigma_{\text{body}}, \sigma_{\text{env}'}, \sigma_{k'}, \Sigma' > \quad \Sigma' = \Sigma \left[ \sigma_{k'} \mapsto \sigma_{e, \sigma_{\text{body}}} \right] \\
 & \text{where } \Sigma(\sigma_{\text{clo}}) = \text{clo}(\sigma_e, \sigma_{\text{env}'}) \quad \Sigma(\sigma_e) = \lambda x. \sigma_{\text{body}} \\
 & \Sigma' \left[ \sigma_{\text{env}''} \mapsto \sigma_{\text{env}} \cdot [x \mapsto \sigma_e] \right]
 \end{aligned}$$

$$20-3/ \quad \langle \sigma_e, \sigma_{\text{env}}, \sigma_k, \Sigma \rangle \xrightarrow{\text{MS}} \langle \sigma'_e, \sigma'_{\text{env}}, \sigma'_k, \Sigma' \rangle$$

$\text{st} =$

$$\text{MS}(\langle \sigma_e, \sigma_{\text{env}}, \sigma_k, \Sigma \rangle) \Rightarrow \langle \sigma_e, \sigma_{\text{env}}, \sigma_k, \Sigma' \rangle$$

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

$\Sigma' = \text{remove All But } (\Sigma, w) = \text{sweep}$

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

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

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

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

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

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

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

$$\Sigma' \quad \text{otherwise}$$

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

$$b \mapsto \emptyset \quad | \quad x \mapsto \emptyset \quad | \quad @ \sigma_L \sigma_R \mapsto \begin{cases} \text{mark}(b) \\ \text{mark}(x) \end{cases}$$

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

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

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

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

20-4 / sound = YES  $n = m \cdot m$

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 vs)