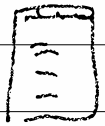
1-1

Goal: Implement a VM

What does a program mean?

"1+1" means "2"

"word.c" means 

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

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

BNF or CFG

$\mathcal{J}_0$: e := v | (+ e e) | (* e e)

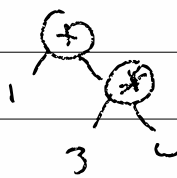
V := number

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

"(+ 1))"

e := v or  or 

V := number



1-2/  = new Add(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 (u big step semantics) : e → v

interp v = v

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

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

1-3/ E^* ← Num* ≈ V*

```
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); }
```

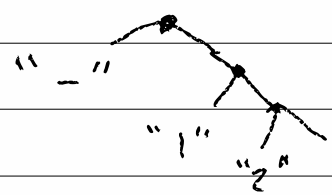
(- e1) = (* -1 e1)

(- e1 e2) = (+ e1 (- e2))

desugar (expander) [compiler] : string tree
string expr
S-expr
Sexpr

Sexpr = string or pair of sexpr or empty

⟹ J_0

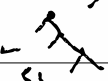


= (- 1 2)

1 - 2

2-1/ Sexpr J_0

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



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

$(+) = 0$

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

$(*) = 1$

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

J_1 e := v | (if e e e) | (e e ...)

v := number | bool | prim

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

interp v = v

interp (if e_c e_t e_e) =

c = interp e_c

e_k = if c e_t else e_f

return interp e_k

interp (e_p e_0 ... e_n) =

p = interp e_p (must be a prim)

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

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

2-2/

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

---

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

- it is partial
- it says nothing about "in between"
- very un-math-like way of proving
- inefficient / unhelpful for implementations

small step : $e \to e$

step (if true e_t e_f) = e_t

step (if false e_t e_f) = e_f

step (p v_0 ... v_n) = $\delta(p, v_0 \ldots v_n)$

if step $e_c = e_c'$ then sstep (if e_c e_t e_f) = (if e_c' e_t e_f)

if step $e_i = e_i'$ then step (e_0 .. e_i e_{i+1} ... e_n) = (e_0 .. e_i' e_{i+1} ... e_n)

2-3)

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

$= (1+1) + 5$

A context, C ::= hole

| if C e e

| if e C e

| if e e C

| (e ... C e ...)

plug : C × e ⇒ e

plug hole e = e

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 × e

2-4/

$e \quad \Rightarrow \quad e$

step $C[\text{if true } e_t \; e_f] = C[e_t]$

step $C[\text{if false } e_t \; e_f] = C[e_f]$

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

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"

$\forall C =$ [redacted]

"quicksort" "mergesort" "heapsort"

"insertionsort"

$\forall l.$ "mergesort l" = "hs l"

$C = ([\text{redacted}] \; L)$

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

time e = e's ans x secs

3-1/ step : e ⇒ e ⇒⇒⇒ v

C[if true e+ ef] → C[et]

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

C := hole | if C e e | if e C e | if e e C |
(e ... C e ...)

evaluation context, E

E := hole | if E e e | (v ... E e ...)
Ehole Eif Eapp

E[if true e+ ec] → E[e+]
E[if false e+ ef] ⇒ E[ef]
E[(p v0 ... vn)] → E[δ(p, v0 ... vn)]

find-redex : e ⇒ (E, e) or #false

find-redex v = false

fr (if e0 e1 e2) = case (fr e0) with
#false ⇒ (hole, (if e0 e1 e2))
(E, e0') ⇒ (if E e1 e2, e0')

3-2/

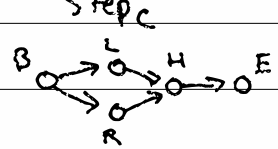
fr (v ... e_0 e_1 ... e_n) =

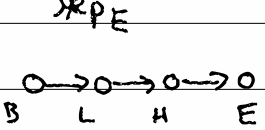
(E, e_0') = fr e_0

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

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

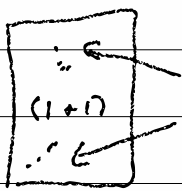
fr (v ...) = (hole, (v ....))





The Standard Reduction theorem

Alonzo Church ((Church-Rosser) Rosser



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

..... (if ... hole ..)^(434) ...))

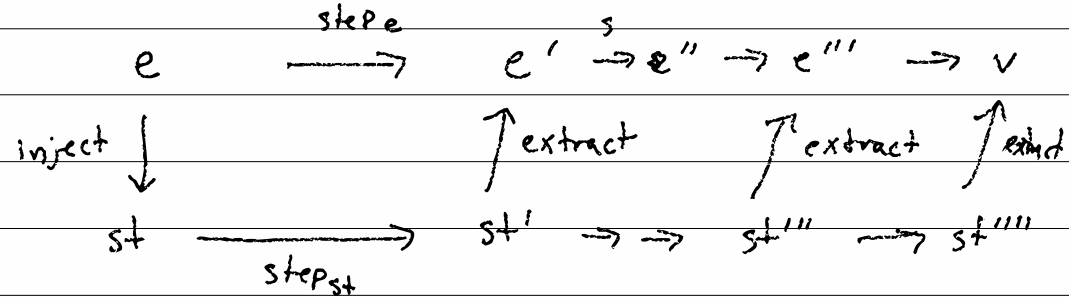
E[(1+1)] --> E [2]

2n time, in space

2n time, in space

3-3/ $step_E : e \Rightarrow e$

machine semantics



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

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

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

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

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

2 $\langle true, E[if\ hole\ e_t\ e_c] \rangle \Rightarrow \langle e_t, E \rangle$

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

4 $\langle e_0\ e_1 \cdots e_n, E \rangle \Rightarrow \langle e_0, E[(hole\ e_1 \cdots e_n)] \rangle$

5 $\langle v, E[v_0 \cdots hole\ e_0\ e_1 \cdots] \rangle \Rightarrow \langle e_0, E[(v_0 \cdots v\ hole\ e_1 \cdots)] \rangle$

6 $\langle v_n, E[p\ v_0 \cdots hole] \rangle \Rightarrow \langle \delta(p, v_0 \cdots v_n), E \rangle$

$(y_n, z_n) \longrightarrow (l_n, l_n)$

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 }
cc-eval e = accv
if (accv != ev) { error }
ll-eval e = allv
if (allv != ev) { error }

3-5/

low level--eval e =

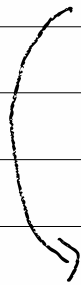
print e as C constructors into "x.c"

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

run "x.bin" and capture output

parse output

return value



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

4-1) $CC_0 \rightarrow CK_0$

$st = \langle e, k \rangle$

$k = kret$ // hole

continuation / kontinuation / stack k

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

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

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

$\Longleftrightarrow \quad B[(v \ldots\ A\ e \ldots)]$

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

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

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

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

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

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

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

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

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

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

(3 4 5)

5-1/ $J_1 \to J_2$

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

$x ::=$ 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 : $e \to e$

J_2 : step : $\Delta \times e \to e$ (with $(f \mapsto d)$)

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

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

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

$x[x \leftarrow v] = v$

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

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

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

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

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

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

```
(define (f x y) (+ (* x 2) (* -1 y)))
(define (g x) (f x x)
(+ 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/ $CK_0 \Rightarrow CK_1$ $st = \langle \Delta, e, k \rangle$

$(1+N)$

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

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

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

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

(if (x > 10 mi) (error) (+ 1 (f x)))

(define (f x) (f x))

(f 0)

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

"Jay"

proper fucton 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

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

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

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

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

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

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

6-2) $CK_1 \Rightarrow CEK_0$

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

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

Jay's envs: $e := make\text{-}v\text{-}empty \mid make\text{-}v\text{-}cons(make\text{-}v\text{-}cons(x, v), env)$

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

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

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

!!! ↓ WRONG ↓ !!! (look ahead ?pgs)

$\langle \Delta, x, mt, k \rangle \mapsto error$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

WRONG:

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

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

Dynamic Scope

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

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

RIGHT

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

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

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

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

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

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

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

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

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

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

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

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

6-5) $J_2 \Rightarrow J_3$

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

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

$x :=$ 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

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

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

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

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

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

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

6-6/

$\langle v_n, \overset{new}{env}, kapp((\lambda\ (x_0 \cdots x_n)\ e)\ v_0 \cdots), (), \overset{old}{env}, k\rangle$
$\mapsto \langle e, mt[x_0 \leftarrow v_0] \cdots [x_n \leftarrow v_n], k\rangle$

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

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

CEK

old $v ::=$ num | bool | prim | $(\lambda\ (x \cdots)\ e)$

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

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

$\langle v_n, —, kapp(($ closure$((\lambda (x_0 \cdots x_n) e), env), v_0 \cdots), (), k\rangle$
$\mapsto \langle e, env[x_0 \leftarrow v_0] \cdots [x_n \leftarrow v_n], k\rangle$

6-7/

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

= ((λ (x0 ... xn) be) e0 ... en)

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

=

((λ (x y) (+ x y))
 5 6)

(let ([x←5])
  (+ x
     (let ([x (+ 1 x)]) (+ x x))
     (+ x 4)))

= (+ 5
     (let ([x←(+ 1 5)]) (+ x x))
     (+ 5 4))

∅[x←5][x←6]

= (+ 5 (+ 6 6) (+ 5 4))

= (+ 5 12 9)

= 26

6-8/ desugar (let* () be) = be

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

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

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

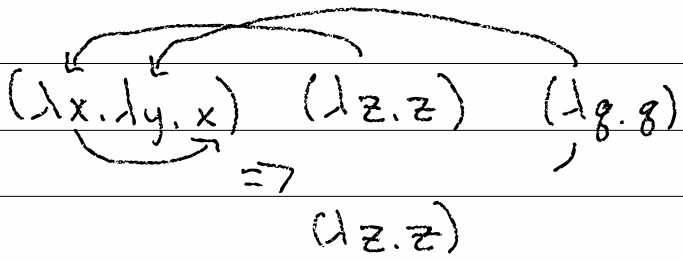
normal let: ((λ (x y z) (+ z z))
  5 (+ x 1) (+ x y))

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

7-1/ ((let [x <- 5])
(λ (y) (+ x y)))
6) => (+ 5 6)

Λ : e = x | λx. e | e e

id : λx. x



Alonzo Church's Lambda Calculus

Church encoding == Object-oriented Prog.

represent as interface interface vs representation

bool : Opt 2 -> Opt 2 -> Some Option 1 or 2

true := λx. λy. x

false := λx. λy. y

if := λb. λt. λf. b t f

if true M N => (λbtf. b t f) true M N

=> true M N => (λxy. x) M N => M

7-2/ Church-encoded numbers

numbers are iteration

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

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

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

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

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

add1 zero ⇒ $\lambda f.\lambda z.\ f\ (zero\ f\ z)$
⇒ $\lambda f.\lambda z.\ f\ z$ = one

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

plus one one ⇒ $\lambda f.\lambda z.$ one f (one f z)
⇒ $\lambda f.\lambda z.$ one f (f z)
⇒ $\lambda f.\lambda z.$ f (f z) = two

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

mult two two ⇒ $\lambda f.\lambda z.$ two (two f) z
$\lambda f.\lambda z.$ (two f) (two f) z
$\lambda f.\lambda z.$ f (f (f (f z))) =
four

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

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

$\Rightarrow T$

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

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

$\Rightarrow F$

fst (pair M N) $\Rightarrow$ M

snd (pair M N) $\Rightarrow$ N

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

fst := $\lambda p.$ p true

snd := $\lambda p.$ p false

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

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

---

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

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

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

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

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

copy := λn. n add1 zero

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

F := λp. pair (snd p) (add1 (snd p))

sub := λn. λm. m sub1 n

fac := λn.

defn ↗ if (zero? n)

one

(mult n (fac (sub1 n)))

copy, not a reference ↙

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)

Ex-5/ $\Omega := \omega\ \omega$

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

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

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

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

$\Rightarrow \omega\ \omega$

$\Rightarrow \Omega$

The Lambda Calculus Σ 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
~~(not~~ (odd? (sub1 x))))
(define (odd? x)
(if (zero? x) false
(even? (sub1 x))))
(even? 10)

Δ = { even? , odd? }

odd? is undefined

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

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

↑ recursive name of fun

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

(desugar (λ (x ...) e)) => (desugar (λ rec (x...) e))

8-2/ OLD:

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

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

NEW:

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

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

$CEK_1 \Rightarrow CEK_2$

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

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

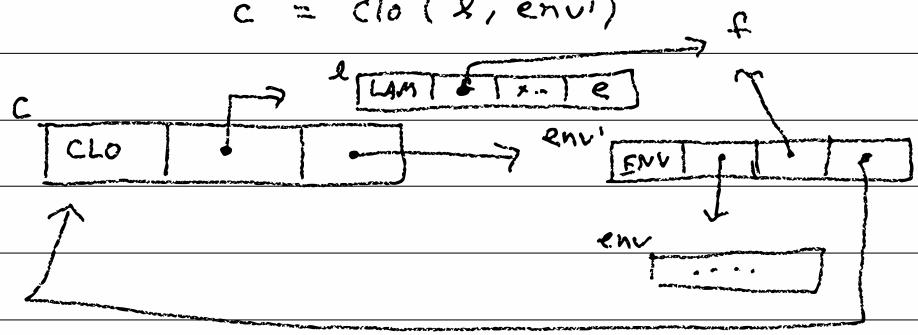
where $\ell = (\lambda\ (x_0 \cdots x_n)\ e)$

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

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

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

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



8-3/ envp = make_env_ext (env, l→f, NULL);
clo = make_clo (l, envp);
envp → val = clo; // ← installs cycle!

```
nat-unfold :=
  (λ rec (f z n)
    (if (= n 0) z
        (f n (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 m C

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

"and"

(pair 1 2)

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

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

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

Variants: "or" List = empty OR node

$v = \ldots \mid (\text{inl } v) \mid (\text{inr } v)$ Unit or (data x list)

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

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

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

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

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

9-2/ CEK ...

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

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

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

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

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

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

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

Bool = Unit or Unit

true = inl ()

false = inr ()

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 x1) ⇒ e1] [(inr x2) ⇒ e2] =

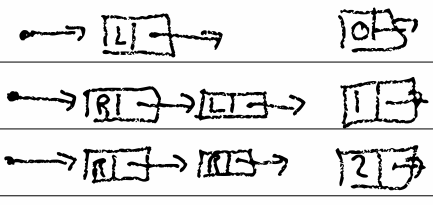
(let ([vs es]) (if (fst vs) (let ([x1 (snd vs)]) e1)
(let ([x2 (snd vs)]) e2)))

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

○ = inl ....

□ = inr (inl ...)

△ = inr (inr ...)



Algebraic Data Types

Type = 1 // Unit tt

0 // Nothing

Type × Type // Pair (pair e1 e2)

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

B // Base types 5

B = Int32 | ...

Bool = 1 + 1

true = inl tt

false = inr tt

Nat = 1 + Nat

0 Zero = inl tt

10 One = inr (inl tt)

110 two = inr (inr (inl tt))

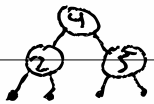
List A = 1 + (A × List A)

mt = inl tt

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

[3, 4] = inr (pair 3 (inr (pair 4 (inl tt))))

Tree A = 1 + (Tree A × A × Tree A)



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

q-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 x K

-> Maybe V

Hash ID (Maybe People) x ID

-> Maybe (Maybe People)

seq : Maybe A x (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

fold f_1 z_1 (fold f_2 z_2 l)

=

fold (λ (a d_{12})

f_1 fst (f_2 a (snd d_{12})) snd)

(z_1, z_2) l

map f_1 (map f_2 l) = map ($f_1 \circ f_2$) l

sum ⇒ sum ↗

Haskell

90-1/

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

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

In C, a variable is a container storing a value

In JS, a variable is a named value

A box is a container that might change

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

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

=subst=>

```
(set-box! (box 7) (+ (unbox (box 7)) 1))
(- (unbox (box 7)) (unbox (box 7)))    => 0
```

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

10-2/

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

Semantics of J_6

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

old $\Rightarrow : e \to e$ $\quad E[(\text{if true } e_t\ e_c)] \Rightarrow E[e_t]$

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

$\Sigma : \sigma \to v$

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

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

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

$\Sigma \times E[\text{set-box! } \sigma\ v] \Rightarrow \Sigma[\sigma \mapsto v] \times E[\overset{v}{\#}]$ $\quad \to$ not c way

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

CESK st = <e, env, sto, k>

sto = mt | sto[σ ↦ v]

<if ec et ef, env, sto, k> ↦ <ec, env, sto, ifk env et ef k>

<v, —, sto, appk (box) () — k>

↦ <σ, mt, sto[σ ↦ v], k>

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

[V_BOX, ptr2]

---

obj_t* delta_setbox (obj_t* args) {

((vbox_t*) second(args)) -> v = first(args);

return make_v_void(); }

10-4/ Option 1:

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

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

Option 2:

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

```
desugar (begin) = (void)    // tt
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/

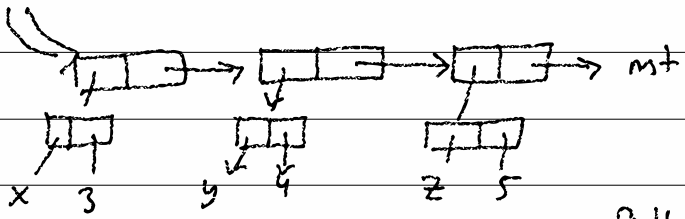
desugar (for [x := e_i ; e_c ; e_f] e_b ...)

= (let ([x e_i])
  (while e_c
    e_b ... e_f))

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

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

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



mtobj = inl tt

field-set o f v =

cons (cons f v) o