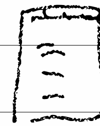
1-1

# Goal: Implement a VM

What does a program mean?

"1+1" means "2"

"wordic" means 

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

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

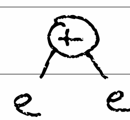
BNF or CFG

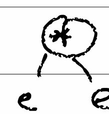
$J_0$: $e := v \mid (+\ e\ e) \mid (*\ e\ e)$

$v := \text{number}$

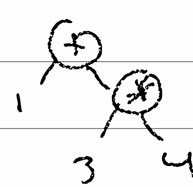
"(+ 1 (* 3 4))" $\in J_0.e$

"(+ 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 ~~lang~~ language M that tells you the semantics of language O

virtual machine: a fast interpreter we like

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

interp $v$ = $v$

interp (+ $e_1$ $e_2$) = (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); }
```

---

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

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

desugar (expander) [compiler] : string tree

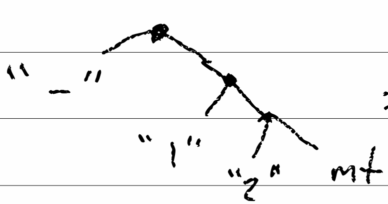
string expr

S-expr

Sexpr

$\Longrightarrow J_0$

Sexpr = string or pair of Sexpr
or empty

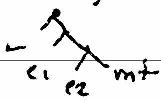


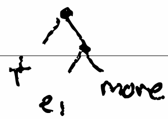

= (- 1 2)

1 - 2

2-1/ Sexpr $\quad$ $J_0$

```
desugar (- e1)         = (* -1 (desugar e1))
        (- e1 e2)      = (+ (d e1) (d (- e2)))
        (+)            = 0
        (+ e1 . more)  = (+ (d e1) (d (+ . more)))
        (*)            = 1
        (* e1 . more)  = (* (d e1) (d (* . more)))
```




---

```
J1  e ::= v | (if e e e) | (e e ... )
    v ::= number | bool | prim
    prim ::= + | * | / | - | ≤ | < | = | > | ≥
```

```
interp v = v
interp (if ec et ef) =
    c = interp ec
    ek = if c et o.w ef
    return interp ek
interp (ep e0 ... en) =
    p = interp ep   (must be a prim)
    v0 ... vn = interp v0 ... interp vn
    ret δ(p, v0 ... vn)
```

2-2

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

(list above v*)

---

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

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

small step : $e \to e$

step (if true et ef) = et

step (if false et ef) = ef

step (p v0 ... vn) = $\delta$(p, v0 ... vn)

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

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

2-3)

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

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

A context, $C$ ::= hole

| if $C$ $e$ $e$

| if $e$ $C$ $e$

| if $e$ $e$ $C$

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

plug : $C \times e \Rightarrow e$

plug hole $e$ = $e$   (plug $C$ $e_p$)

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 : $e \Rightarrow C \times e$

2-4/

$e \Rightarrow e$

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

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

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 "

? C = [illegible]

" quicksort" " mergesort" "heapsort"

"insertion sort"

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

C=( [illegible] L)

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

time e = e'sans × secs

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

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

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

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

evaluation context, E

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

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

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

find-redex v = false

fr (if $e_0\ e_1\ e_2$) = case (fr $e_0$) with
  #false $\Rightarrow$ (hole, (if $e_0\ e_1\ e_2$))
  $(E, e_0') \Rightarrow$ (if $E\ e_1\ e_2$, $e_0'$)

3-21

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

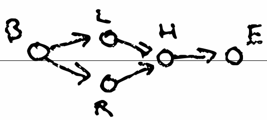
  (E, $e_0'$) = fr $e_0$

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

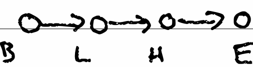
  new EApp(v, E, $e_1$ .. $e_n$)

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

step$_C$



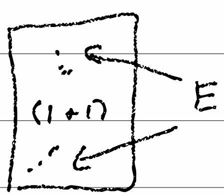

step$_E$




The Standard Reduction theorem

Alonzo Church ((Church-Rosser)) Rosser

---




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

  ..... (if .... hole ...)(+34) ...))

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

in space    in space

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

machine semantics

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

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

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

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

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

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

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

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

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

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

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

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

3-4/

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

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

```
test ( se, ex-val ) ==
    e = desugar se
    big-step eval (e) = actual-big-val
    if (abv != ev) { error }
    small-step eval (e) = ac - sm - v
    (if (asv != ev) { error }
    cc0-eval e = accv
    if (accv != ev) { error }
    ll-eval e = allv
    if (allv != ev) { error }
```

3-5

low level-eval e =

print e as C constructors into "x.c"

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

run "x.bin" and capture output

parse output

return value

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

9-11 $\quad CC_0 \rightarrow CK_0$

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

$k$ = kret // hole

continuation kontinuation stac k

kif e e k // if E e e

kapp (v...) (e...) k // (v... E e...)

$$A \rightarrow \text{kapp}\ (v \ldots)\ (e \ldots)\ B$$

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

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

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

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

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

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

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

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

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

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

$$\delta(\text{SUB}, (v_1\ v_0)) = v_0 - v_1$$

$$(3\ 4\ 5)$$