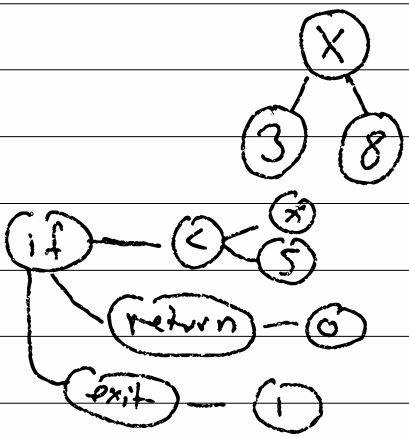
1-1/ 1 + 1 5 1+

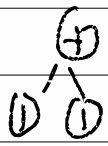
1 x 3 "3 x 8"



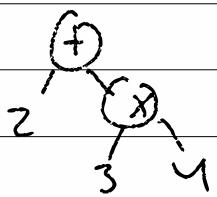

```
if (x < 5) {
    return 0; }
else {
    exit(1); }
```

(+ 1 1)

bp children



(+ 2 (x 3 4))

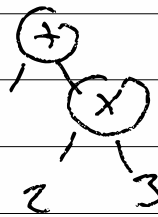


1-2) $J_0 \Rightarrow$ e := v | (+ e e)
v := number | (* e e)

(+ 1 (+ 2 3)) ∈ J_0

```
interface Joe { }
class JNumber implements Joe {
  int n;  JNumber(int _n) { n = _n; }}
class JPlus imp Joe {
  Joe left, right;  JPlus(...) }
class JMult imp Joe {
  Joe l, r;  JMult() ... {} }
```

(+ 1 (* 2 3)) → Sexpr =




```
new JPlus(
  new JNum(1),
  new JMult(
    new JNum(2))
    new JNum(3)))
```

= JP(JN(1), JM(JN(2), JN(3)))

```
class JPlus:
  def _init(l, r):
    this.l = l;
    this.r = r;
```

BST n := mt | (br num n n)

1-3/6 pp : J_0 => string

① pp n = itos (n)

② pp (+ e_L e_R) = "(" ++ pp(e_L) ++ "+" ++ pp(e_R) ++ ")"

③ pp (* e_L e_R) = "(" ++ pp(e_L) ++ "*" ++ pp(e_R) ++ ")"

① interface J0e { public String pp(); }

② class JNum { ...
    public String pp() {
        return intToStr(n); } }

③ class JPlus {
    public String pp() {
        return this.left.pp() ++ "+" + this.right.pp(); }

1-4/ big-step interpreter

```
interp : e -> v

interp n = n
interp (+ eL eR) = interp eL + interp eR
interp (* eL eR) = interp eL * interp eR
```

(1)

```
class JMult {
  public int interp() {
    return this.left.interp() * this.right.interp(); }}
```

```
(+ 1 2 3) = (+ 1 (+ 2 3))
```

desugar →

```
se = empty | (cons se se) | string
(a b c) = (pair "a" (pair "b" (pair "c" mt)))
(+ 1 2) = (p "+" (p "1" (p "2" mt)))
(+ 1 (+ 2 3)) = (p "+" (p "1"
                  (p (p "+" (p "2"
                            (p "3" mt)))
                     mt)))
```

1-5/ desugar for J0

("-" e) => (* -1 (desugar e))

("-" e1 e2) => (+ (d e1) (de ("-" e2)))

("+") => 0

("+" e1 more ...) =>

(+ (d e1) (d ("+" more ...)))

'*'

"x"

(x

= 1

"x"

se => J0 => v

desugar interp

compile bc => v

vm

2-11

desugarer

$(- \ e_1) \Rightarrow (\hat{*} \ -1 \ e_1')$

$(- \ e_1 \ e_2) \Rightarrow (\hat{+} \ e_1' \ (- \ e_2))$

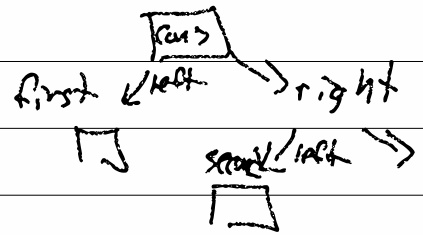
$(+) \Rightarrow 0$

$(+ \ e_1 \ e_2 \ \ldots) \Rightarrow (\hat{+} \ e_1' \ (+ \ e_2 \ \ldots))$

```
def
desugar(se):
    if isList(se) && length(se) = 2 &&
        first(se) == "-" then
```

```
> def length(se):
>     if isNull(se): return 0
>     else if cons(se): return 1 + length(rightTo)
>     else false
```

```
        new JMult(new JNum(-1), desugar(
                                  second(se)))
```




```
if isList(se) && len(se) = 3 && first(se) == "-"
then: return new JAdd(desugar(second(se)),
                      desugar(new cons("-",
                                       new cons(third(se), null))
```

22) cons(a, cons(b, cons(c, null))))

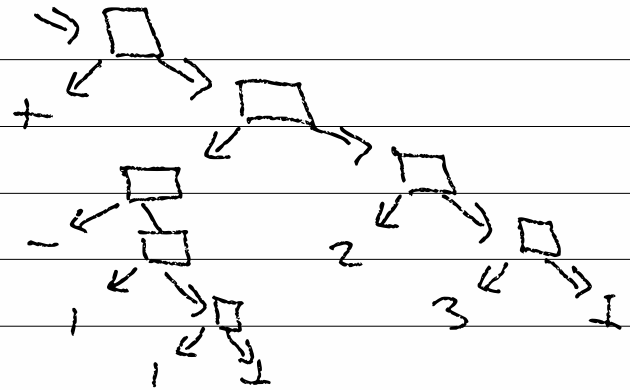
len = 3

(+ (- 1 1) len = 4

1 2 2 ← 3

3)

4



isList(se):

Null: True

cons: isList(right(se))

ow: False

Len(se):

Null: 0

cons: 1 + len(right(se))

2-3/ $J_0 \to J_1$

fun, args

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

c, t, f

$v ::= b$

$b ::=$ some set of constants

// in J_0, b = num | + | * | numbers | bools | prim

prim = +, -, *, /, ≤, <, =, >, ≥, ...

interp $v = v$

interp (if e_c e_t e_f) = interp e_k

where e_k = if interp e_c then e_t o.w. e_f

interp (e_f e_a ...) = $\delta(p, v_a \ldots)$

where p = interp e_f

v_a.. = interp e_a ..

$\delta : b \ldots \to b$

$\delta(+, 1, 2) = 3$ $\quad \delta(/, 1, 0) = \perp$

$\delta(<, 1, 3) = \text{true}$

24/ "small step interp" "big step"

e ⇒ e e ⇒ v

until its the same

interp Interp

```
Interp e =
  let e' = interp(e)
  if e == e' then
     ret e
  o.w.
     Interp (e')
```

(+ (+ 1 1) 2) ⇐ (+ (+ 1 1) (+ 1 1))

⇒ (+ 2 2)

⇒ 4

(+ 2 (+ 1 1))

```
int x = 1;
f ( x--, x++)      (1, 0)
                   (2, 1)
```

2-5/ step : e → e

```
step (if true  et ef) = et
step (if false et ef) = ef
step (p va ...) = δ(p, va ...)
step v = v
step (if e (&v) et ef) =
  (if (step e) et ef) or (if e (step et) ef)
step (vb ... e(&v) ea ...) =
  (vb ... (step e) ea ...)
```

A context

```
C ::= hole | if0 C e e
           | if1 e C e
           | if2 e e C
           | (e ... C e ...)
```

plug C e (C[e])

```
plug hole x = x
plug (if0 C e1 e2) x = if x e1 e2
plug (if1 e1 C e2) x = if e1 x e2
plug (e1 ... C e2 ..) x = (e1 ... x e2 ...)
```

2-6

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 $C[p\ v_a \ldots] = C[\delta(p, v_a \ldots)]$

"parse" : $e \Rightarrow C \times e$

step $\Longrightarrow e$

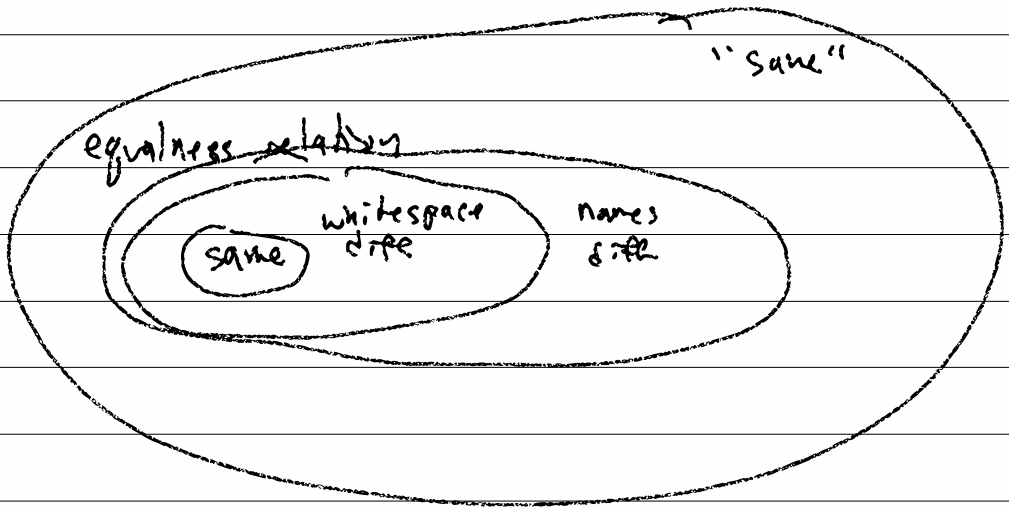
```
iStep e = if e ∈ V then e
  C, e' = parse e e
  e''   = step e'
  plug C e''
```

redex

```
parse : e => C x e
parse (if ec et ef) =
  if ec ∈ V then (hole, e)
  o.w. let C', e' = parse ec
       (ifO C' et ef , e')
```

2-7 Answer: Contexts

Question: How do I know when two programs do the same thing?



$x = y$

$\forall x.\ fx = gx$

$\forall C.\ C[x] = C[y]$

C = hole x = y

C = (+ hole 2) x+2 = y+2

C = (map hole (list 1 2))

....

Observational Equivalence

2-8) $C ::= \text{hole} \mid \text{if } C\ e\ e \mid \text{if } e\ C\ e \mid \text{if } e\ e\ C \mid (e \ldots C\ e \ldots)$

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

"unique decomposition" ↓ unique E

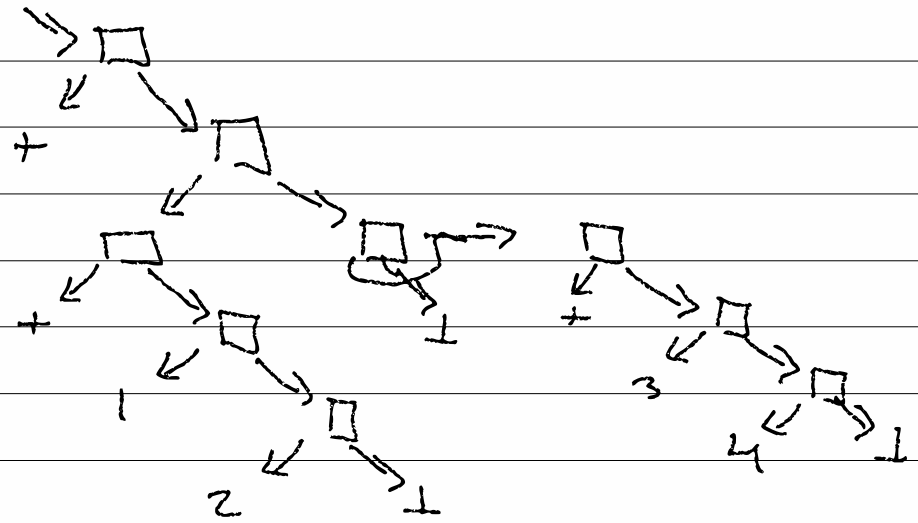
$\forall e.\ e \in v$ or $e = E[e']$ where $e' \notin v$

$\delta(1, 2) = 1 \qquad \delta(1, 1, 0) = 2$

$(+\ (+\ 1\ 2)\ (+\ 3\ 4))$ = tree = java tree

```
new JPlus (new JPlus (new JNum (1),
                      new JNum (2))
           new JPlus (new JNum (3),
                      new JNum (4)))
```

2-9



(+ (+ 1 2)
(+ 3 4))

3-1/ E = hole | (if E e e)
| (v ... E e ...)

step E[if true e+ ef] → E[e+]
step E[if false e+ ef] → E[ef]
step E[p va ...] → E[δ(p, va ...)]

interp e = case (parse e) of
false ⇒ e
(E, e) ⇒ e' = step e
E[e']

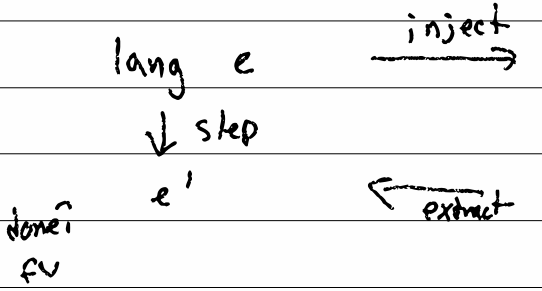
gigantic program [(+ (+ 1 1)
(+ 2 3))]

3-2/ Sp "language"

CC$_0$ "machine"

$e \to e$

$st \mapsto st$



$st = \langle e, E \rangle$ done? $\langle v, hole \rangle$

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

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

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

$\langle true, E[if\ hole\ e_t\ e_f] \rangle \mapsto \langle e_t, E \rangle$.

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

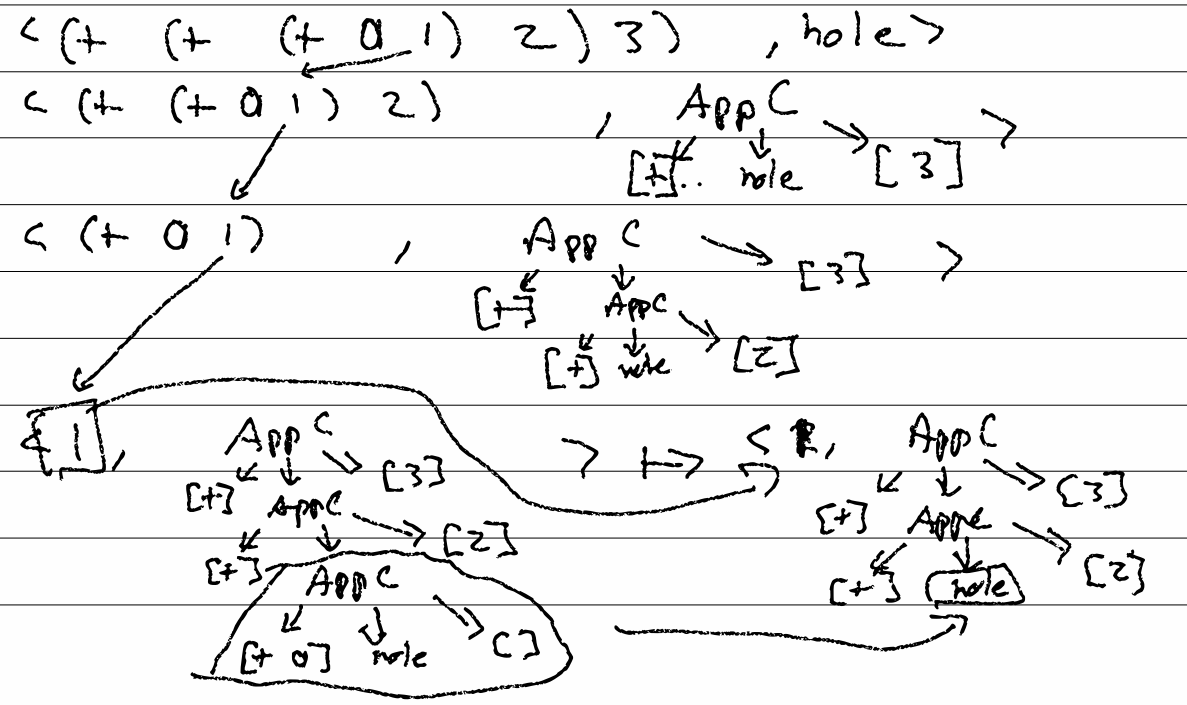
$\langle e_0\ e_1 \dots, E \rangle \mapsto \langle e_0, E[hole\ e_1 \dots] \rangle$

$\langle v, E[v_0 \dots hole\ e_1\ e_2 \dots] \rangle \mapsto \langle e_1, E[v_0 \dots v\ hole\ e_2 \dots] \rangle$

$\langle v_n, E[v_0 \dots hole] \rangle \mapsto \langle \delta(v_0 \dots v_n), E \rangle$

3) E = hole | if E e e | (v ... E e ...)

```
interface Context {
    Expr plug(Expr); }
Hole : Context {
    plug(e) = e; }
IfC : Context {
    Context c; Expr t, f;
    plug(e) = new If(c.plug(e), t, f); }
AppC : Context {
    List<V> vs; Context c; List<Expr> es;
    plug(e) = new App(vs ++ [c.plug(e)] ++ es); }
```



3-4// E = hole | if E e e | $v \ldots E\ e \ldots$

= top | if e e $\square$ | $(v \ldots)(e \sim) \square$

K = kret | kif e e K | kapp $\vec{v}$ $\vec{e}$ k

CK$_0$ machine $st = \langle e, K \rangle$

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

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

$\langle e, \text{kif } e_t\ e_f\ K \rangle$ = extract $\langle \text{if } e\ e_t\ e_f, K \rangle$

$\langle e, \text{kapp } (v \ldots)(e_1 \ldots) K \rangle$ = extract $\langle (v \ldots e\ e_1 \ldots), K \rangle$

done $\langle v, \text{kret} \rangle$

0 $\langle \text{if } e_c\ e_t\ e_f, K \rangle \mapsto \langle e_c, \text{kif } e_t\ e_f\ K \rangle$

1 $\langle \text{true}, \text{kif } e_t\ e_f\ K \rangle \mapsto \langle e_t, K \rangle$

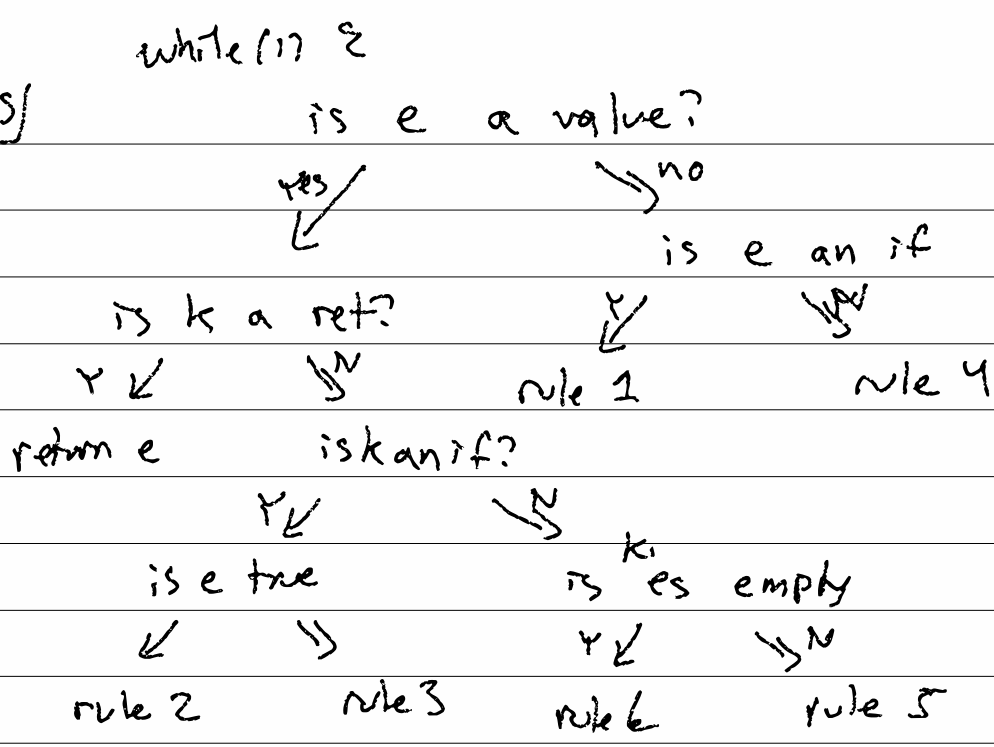
2 $\langle \text{false}, \text{kif } e_t\ e_f\ K \rangle \mapsto \langle e_f, K \rangle$

4 $\langle e_0\ e_1 \ldots, K \rangle \mapsto \langle e_0, \text{kapp } ()\ (e_1 \ldots)\ K \rangle$

5 $\langle v_1, \text{kapp } (v_0 \ldots)\ (e_0\ e_1 \ldots)\ K \rangle$
$\mapsto \langle e_0, \text{kapp } (v_0 \ldots v_1)\ (e_1 \ldots)\ K \rangle$

6 $\langle v_n, \text{kapp } (v_0 \ldots)\ ()\ K \rangle$
$\mapsto \langle \delta(v_0 \ldots v_n), K \rangle$

3-5)



rule 1:

k = new kif (e.true, e.false, k)

e = e.cond;

~~jump R.PC~~

k is a stack and the stack (of c) = kontinuation

continuation

3-6/

```
struct if {
  expr h;
  expr *c, *t, *f; }

struct num {
  expr h;
  int n; }

struct app {
  expr h;
  expr *f, *args; }

expr *make_if(expr *c, t, f){
  if *p = malloc(sizeof ...))
  p->h.tag = IF;
  p->c = c; ...
  return p; }
```

```
struct expr {
  enum tag; }

enum tag {
  IF, NUM, APP,
  BOOL, PRIM,
  KRET, KIF,
  KAPP, CONS, NIL};
```

(+ (+ 1 1) 7 2)

```
make_add ( make_add (make_num(1),
                     make_num(1)),
           make_num(2));
```

0201122207