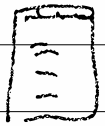
1-1

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)

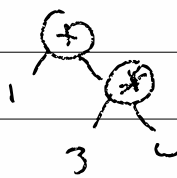
V := number

"(+ 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); }
```

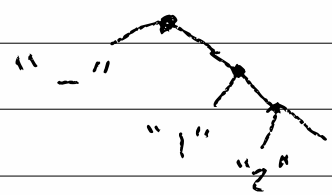
(- e1) = (* -1 e1)

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

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

Sexpr = string or pair of sexpr or empty

⟹ J_0

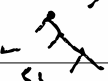


= (- 1 2)

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/

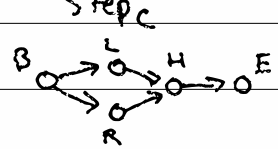
fr (v ... e_0 e_1 ... e_n) =

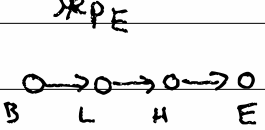
(E, e_0') = fr e_0

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

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

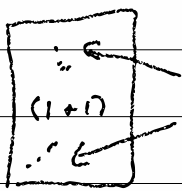
fr (v ...) = (hole, (v ....))





The Standard Reduction theorem

Alonzo Church ((Church-Rosser) Rosser



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

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

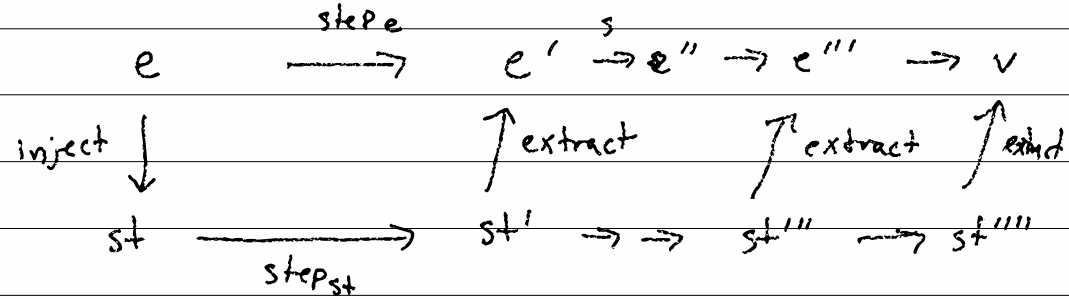
E[(1+1)] --> E [2]

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 =

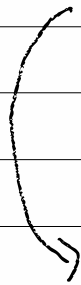
print e as C constructors into "x.c"

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)))