2-1/

(+ (read) (+ 12 (- 10)))

R_1 : e ::= .... | var | (let ([var e]) e)
(let var x = e in e)

(let x := 1 in
(let y := 2 in => 3
(+ x y)))

(let x := 1 in
(let x := 2 in => 4
(+ x x)))

(let x := (read) in
(+ x x)) => ???

2-2)

R_0 : $interp_e : e \Rightarrow num$ $\quad interp_p : p \Rightarrow num$

R_1 : $interp_e : env \rightarrow e \Rightarrow num$ $\quad interp_p : p \Rightarrow num$

env : $(var \rightarrow num)$

```
interp env e = case e of
  Num n -> n
  Neg e -> -1 * interp env e
  Add l r -> (interp env l) + (interp env r)
  Let x xe be -> interp env' be
      where env' = env [x |-> interp env xe]
  Var x -> env x
```

2-3)

randp : int → R_0

randp : set(var) × int → R_1 ← are the bound variables

randp vs 0 = choices

randp vs (1+n) = choices

random num (read)

v ← vs

(- (randp vs n))

(+ (randp vs n) (randp vs n))

(let x := randp vs n in
randp vs' n)

where vs' = vs ∪ {x} x is a random variable

2-4)

(let z0 := (let v1 := z in (+ v1 3))
(- (+ v0 (read))))

⇒ (+ -5 (read))

opt : env × e → e

opt env (Var x) = env x

opt env (Let x xe be) =
let xe' := opt env xe in
if simple? xe' then
opt env [x → xe'] be
o.w.
(Let x xe' (opt env [x→x] be))

simple? : e ⇒ bool

Var, Num ⇒ T
o.w F

2-5)

(let x := (+ (read) (read)) in
  (+ 2 x))
  (1) (2)
⇒ (+ 2 (+ R R))

(+ x x)
↛ (+ (+ R R) (+ R R))

(+ (read) x)
  (3)
⇒ (+ R (+ R R))
     1   2 3

2-6/ X86-64 assembly

compile : $R_1 \Rightarrow X_0$

X_0 : p ::= (program info [label ↦ blk ...] ...)

blk ::= (block info instr ...)

instr ::= (addq arg arg) | (subq arg arg) | (movq arg arg) | (retq) | (negq arg) | (callq label) | (jmp label) | (pushq arg) | (popq arg)

arg ::= $n | %rn | offset(%rn) | %rn (offset) | var

rn ::= rsp, rbp, rax, rbx, rcx, rdx, rsi, rdi, r8 ⇒ r15

2-7/

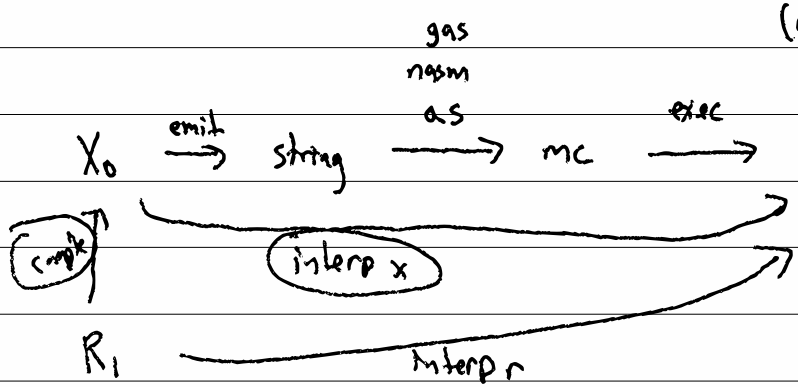
emit : X => outout

emit (program _ blks) :=
".globl main"
"main:" emit (blks)
emit [label ⇒ (block _ mshs)] :=
"%label:" emit (mshas)
emit (addq src dst) = "addq" emit(src), emit(dst)
emit (constant n) = "$" n

2-8

```
.globl main
main:
    movq  %rax, $8
    movq  %rbx, $10
    addq  %rax, %rbx
    retq
```

```
(program -
 [main ->
  (block -
   [(movq (reg rax) (con 8))
    (movq (reg rbx) (con 10))
    (addq (reg rax) (reg rbx))
    (retq)])])
```



2-9)

$\chi_{i\,p} : \chi_{0,p} \Rightarrow ms$

$\chi_{i\,p}\ (program - lab \Rightarrow blk) := \chi_{i\,b}\ ms_0\ \text{"\_main"}$

$ms := (nn \Rightarrow num) \times (addr \Rightarrow num) \times (var \Rightarrow num) \times (lab \Rightarrow blk)$

$ms_0 = (\lambda m.\ 0) \times (\lambda addr.\ 0) \times (\lambda v.\ 0) \times lab \Rightarrow blk$

$\chi_{i\,b} : ms \times lab \Rightarrow ms$

$\chi_{i\,b}\ ms\ lab = \chi_{i\,s}\ ms\ (ms, lab \Rightarrow blk\ lab), instrs$

$\chi_{i\,s} : ms \times List(instr) \Rightarrow ms$

$\chi_{i\,s}\ ms\ \emptyset = ms \qquad \chi_{i\,s}\ ms\ (cons\ i_0\ ir) = \chi_{i\,i}\ ms\ i_0\ ir$

2-10

xii : ms x instr x List(instr)

xii ms (addq src dst) k = xis ms' k
  where ms' = ms [dst ↦ ms(src) + ms(dst)]

ms [arg ⇒ nn num] := case arg of
  Constant _ → error
  Reg rn ⇒ ms { rn⇒num = rn⇒num [rn ↦ nn] }
  Var x ⇒ ms { v⇒num = v⇒num [x ↦ nn] }
  Deref rn off ⇒ ms { mem = mem [ms(Reg rn) + off ⇒ nn] }

2-11/

ms [arg] : num

ms [Constant n] = n

ms [Reg rn] = ms.regs (rn)

ms [Var x] = ms.vars (x)

ms [Offset rn off) = ms.mem (ms[Reg rn] + off)

xii ms (pushq src) k = xis ms' k

ms' = ms [%rsp (0) ⟼ ms (src)
%rsp ↦ ms (%rsp) - 8]

xii ms (popq dst) k = xis ms' k

ms' = ms [dst ↦ ms (%rsp(0)), %rsp ↦ ms(%rsp) +8]

2-12

xii ms (jmp lab) k = xi b ms lab

xii ms (callq "_read") k = λiß ms' k

ms' = ms [%rax ↦ do-a-read]

xii ms (retq) k = escape from xi

and return ms(%rax)