2-3 | 4 | assign-homes | X^* | X^0

job: map variables to memory (on the stack)

$X^0 = X$ (but may break some x86-rules)

frame setup +

```
[ movq  $3, !body      ]  =>  [ movq  $3, -8(%rbp)  ]
[ addq  $5, !body      ]     [ addq  $5, -8(%rbp)  ]
```

+ frame cleanup

```
[ movq  $3, !body      ]  =>  [ movq  $3, -8(%rbp)  ]
[ movq  $8, !lhs       ]     [ movq  $8, -16(%rbp) ]
```

(calculate the frame size)

5 | patch | X^0 | X

X^*

```
[ movq  !lhs, !rhs ]  =>  [ movq  -8(%rbp), -16(%rbp) ]
```

```
[ movq  -8, %rax  ]
[ movq  %rax, -16 ]
```

3-1

parse : str → R_1

---

compile : R_1 → assembly (str)

1. uniquify : R_1 → R_1
2. flatten : R_1 → C_0
3. select-instr : C_0 → X^* (asm but w/ variables)
4. assign-homes : X^* → X° (asm but breaks rules)
5. patch-inst : X° → X (asm but a tree)
6. print : X → assembly (str)

---

as : asm → bin object

ld : obj* → exe

---

1. uniquify

```
(let ([x 5])
  (let ([x 7])
    (+ x x)))
```

⇒

```
(let ([x1 5])
  (let ([x2 7])
    (+ x2 x2)))
```

maintain counter
inc on newlet
on newlet append counter to variable
on
- add [old ↦ new] to the map
- a var reference, look at map
- when you leave the let body, use original map

map(old name → new name)

map.add : map key value → new map

```
(let ([x 5])
  (+
    (let ([x 7])
      (+ x x))
    x))
```

⇒

```
(let ([x1 5])
  (+
    (let ([x2 7])
      (+ x2 x2))
    x1))
```

3-2

flatten : $R_1 \Rightarrow C_0$

int | read | (- R_1) | (+ R_1 R_1)

flatten (int) = (program () (ret int))

flatten (the-var , int) = (program (the-var) (:= the-var int))

(read) = (program (rr) (:= rr (read)) (ret rr))

(tv , read) = (program (tv) (:= tv (read))

((- e)) = with (e) = (program (vs) inst... (ret a

(program (ans ∪ vs)

inst ... (:= ans (- a)) (ret ans))

(tv , (- e)) = with ~~alet = (progra~~ o(tv , e) =

(program (vs) (program (vs) inst...)

inst ... (:= tv (- tv)))

((+ l r)) = with (l) = VS_L $inst_L$ ... (ret a_L)

(r) = VS_R $inst_R$ ... (ret a_R)

(program (ans ∪ VS_L ∪ VS_R)

$inst_L$ ... $inst_R$ ... (ans := (+ a_L a_R)) (ret ans))

o (tv (+ l r)) = with o(new , l) = VS_L $inst_L$

o(tv , r) = VS_R $inst_R$

(program (new ∪ VS_L ∪ VS_R)

$inst_L$ ... $inst_R$ ... (tv := (+ new tv)))

(x) = (program (x) (ret x))

(tv , x) = (program (x tv) (:= tv x))

3-3

[redacted] ((let ([x xe]) be)) =

[redacted](xe) = vsxe instxe (ret axe)

[redacted](be) = vsbe instbe (ret abe)

(program (x ∪ vsxe ∪ vsbe)
instxe ...
(:= x axe)
instbe ...
ret abe)

o(tv, (let ([x xe]) be)) =

o(x, xe) = vsxe instxe

o(tv, be) = vsbe instbe

(program (x ∪ vsxe ∪ vsbe)
instxe ... instbe ...)

(:= x y)
(ret y)

⇓

[redacted]
(ret x)

opt : $C_0 \to C_0$

(+ (+ 1 2) 3)

⇓

(x = (+ 1 2)
ret (+ x 3))

select-inst : $C_0 \to X^*$

si ((:= x (+ 1 r))) = addq 1, r, x

movq [redacted] 1, x
addq r, x

movq r, (x)
addq 1, (x)

si ((:= x (+ 1 x)) = addq 1, (x)

si ((:= x (read)) = callq read_int
movq %rax, (x)

si ((ret a)) = movq a, %rax

3-4 assign-homes : $X^* \to X^0$

(program (vs ...) (x86-inst ...) is ... ret)

let k = |vs ...|

count = k if even and k+1 if odd

vs = (x y z)

k = 3

count = 4

[pushq %rbp movq rsi, rbp
  subq count, rsi] setup

is' ... = map rename(σ) is

σ = [x ↦ 0
     y ↦ 1
     z ↦ 2]

[addq count, rsi
  popq rbp] restore

ret

rename (σ, addq x, y)
= addq -x⃗*8(%rbp) σ(x) = x⃗
       -y⃗*8(%rbp) σ(y) = y⃗

---

patch : $X^0 \to X$

movq x, y
⇒

movq -8(%rbp), -16(%rbp)

⇒

movq -8(%rbp), %rax
movq %rax, -16(%rbp)

mun : inst -> list(inst)

mut : inst (inst -> void) -> void