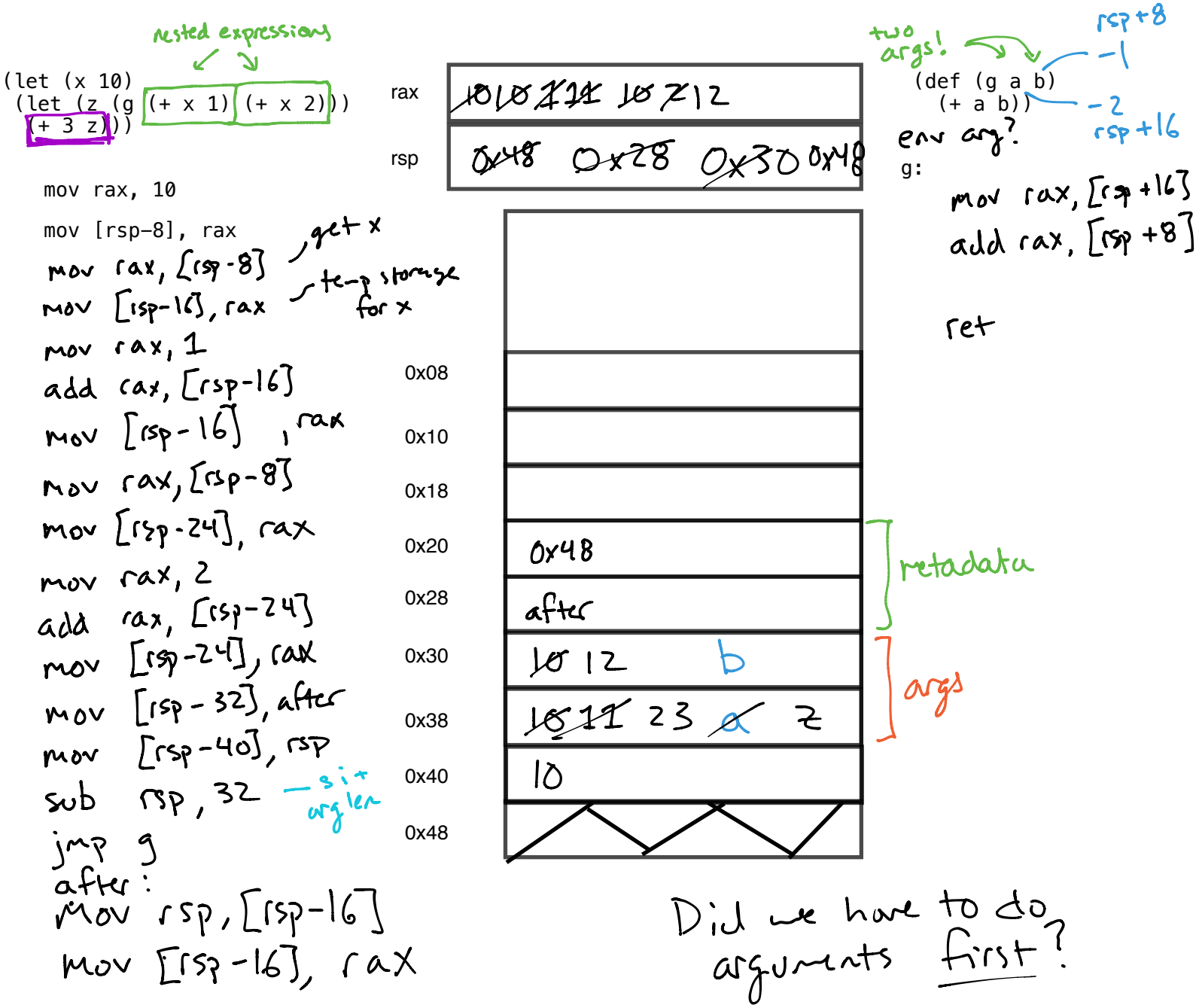
Calling convention for one argument:

Call setup (one arg version):
- Move return address, then current rsp, then argument 1, then arg 2
- Always start at current si for return address, count up
- Subtract to point rsp at the return address

Callee (one arg version):
- Rely on (first) argument in [rsp-16], so env starts with [(arg, 2)] [rsp-24]
- Start at a "higher" si=3 for any local vars
- Expect [rsp] to contain return pointer, use ret

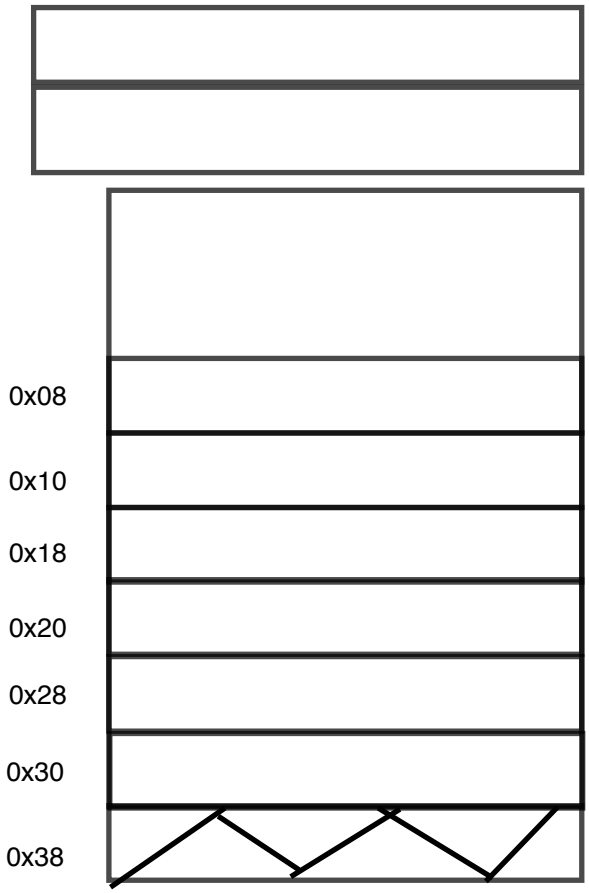
After the call (one arg version):
- Rely on old rsp at [rsp-16] (a true constant)
- Expect answer to be in rax from callee

What about two arguments?



```
(def (twosqrt3x x)
 (let (ans (do_sqrt (* x 3)))
  (* 2 ans)))
(twosqrt3x 4)
```

```
int do_sqrt(int val) {
  float asF = (float)val;
  return (int)(sqrt(asF));
}
```




```
   twosqrt3x:
      mov rax, [rsp-16]
      imul rax, 3
      sub rsp, 16
      mov rdi, rax
      call do_sqrt
0x900 mov rax, 4
      add rsp, 16
      mov [rsp-24], rax
      mov rax, 2
      imul rax, [rsp-24]
      ret
   our_code_starts_here:
      mov [rsp-8], after_call
      mov [rsp-16], rsp
      mov [rsp-24], rax
      sub rsp, 8
      jmp twosqrt3x
      after_call:
      ret
```

(let (x 3) (set x 10) x)

```
(def (fact n)
  (if (< n 2) 1
      (* n (fact (- n 1)))))
(fact 3)
```

work to do w/ ret value

(let (n 3)
  (if (< n 2) 1
      (* n (fact (- n 1)))))

(many steps)

(let (n 3)
  (* 3 (let (n 2)
         (* 2 (let (n 1)
                1))))))

after_call: (3)
imul rax, [rsp-16]
ret

after_call: (2)
imul rax, [rsp-16]
ret

mov rax, 1
ret

accumulator

```
(def (fact n sofar)
  (if (< n 2) sofar
      (fact (- n 1) (* n sofar))))
(fact 3 1)
```

(let ((n 3) (sofar 1))
  (if (< n 2) sofar
      (fact (- n 1) (* n sofar)))))

no work to do w/ ret value

(let ((n 3) (sofar 1))
  (let ((n 2) (sofar 3))
    (let ((n 1) (sofar 6))
      sofar)))

after_call: ret

after_call: ret

mov rax, [rsp-16]
ret