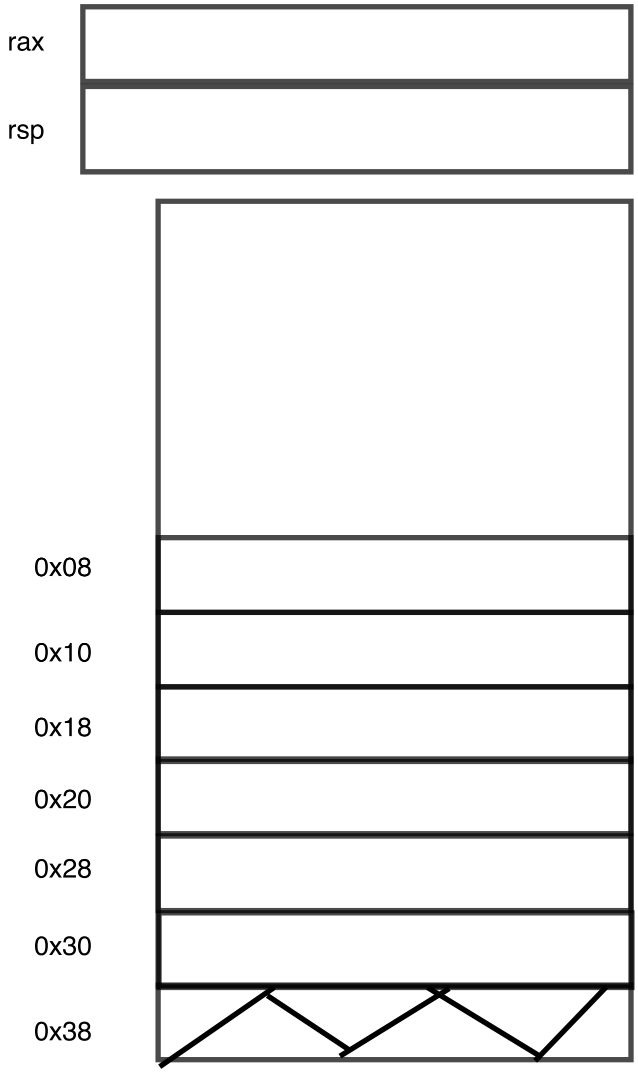
```
type expr =
  | EApp of string * expr

type def =
  | DFun of string * string * expr

type prog = def list * expr

type prog = def list * expr
let parse_def (sexp : Sexp.t) : def = ... parse ...
let parse_program (sexps : Sexp.t list) : prog = ... parse ...

let rec e_to_is (e : expr) (si : int) (env : tenv) (defs : def list) : string list =
... other cases as before ...
| EApp(f, arg) ->
```

```
let compile (program : Sexp.t list) : string =
  let (defs, body) = parse_program program in
  let instrs = e_to_is body 1 [] defs in
```

```
(let (x 10)
 (let (z (g x))
  (+ 3 z)))
```

```
mov rax, 10
mov [rsp-8], rax
   mov rax, [rsp-8]
   mov [rsp-16], after_call
   mov [rsp-24], rsp
   mov [rsp-32], rax
   sub rsp, 16
   jmp g
   after_call:
mov rsp, [rsp-16]
mov [rsp-16], rax
mov rax, 3
add rax, [rsp-16]
```



```
(def (g y)
  (+ y 1))

g:
  mov rax, [esp-16]
  add rax, 1
  ret
```

One possible calling convention, but not the only one possible!

Call setup:

- Move return address, then current rsp, then argument
- Always start at current si for return address, count up
- Subtract to point rsp at the return address

Callee:

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

After the call:

- Rely on old rsp at [rsp-16] (a true constant)
- Expect answer to be in rax from callee