```
type op =
  | Inc
  | Dec

type expr =
  | ENum of int
  | EOp of op * expr
  | EId of string
  | ELet of string * expr * expr

let int_of_string_opt s =
  try
    Some(int_of_string s)
  with
    Failure _ -> None

let rec sexp_to_expr (se : Sexp.t) : expr =
  match se with
    | Atom(s) ->
      (match int_of_string_opt s with
        | None -> EId(s)
        | Some(i) -> ENum(i))
    | List(sexps) ->
      match sexps with
        | [Atom("inc"); arg] -> EOp(Inc, sexp_to_expr arg)
        | [Atom("dec"); arg] -> EOp(Dec, sexp_to_expr arg)



        | _ -> failwith "Parse error"
let parse (s : string) : expr =
  sexp_to_expr (Sexp.of_string s)
```

put the answer in rax!

```
open Printf

let stackloc i = (i * 8)

type tenv = (string * int) list

let rec find (env : tenv) (x : string) : int option =
  match env with
    | [] -> None
    | (y, i)::rest ->
      if y = x then Some(i) else find rest x

let rec expr_to_instrs (e : expr) (si : int) (env : tenv) =
  match e with
    | EId(x) ->
      (match find env x with
        | None -> failwith "Unbound id"
        | Some(i) ->
          [sprintf "mov rax, [rsp - %d]" (stackloc i)])
    | ELet(x, value, body) ->
      let v_is = e_to_is value si env in
      let store = "mov [rsp - %d], rax" (stackloc si)
      let b_is = e_to_is body (si + 1) ((x,si)::env) in
      v_is @ [store] @ b_is
    | ENum(i) -> [sprintf "mov rax, %d" i]
    | EOp(op, e) ->
      let arg_exprs = expr_to_instrs e si env in
      match op with
        | Inc -> arg_exprs @ ["add rax, 1"]
        | Dec -> arg_exprs @ ["sub rax, 1"]
```

stack index

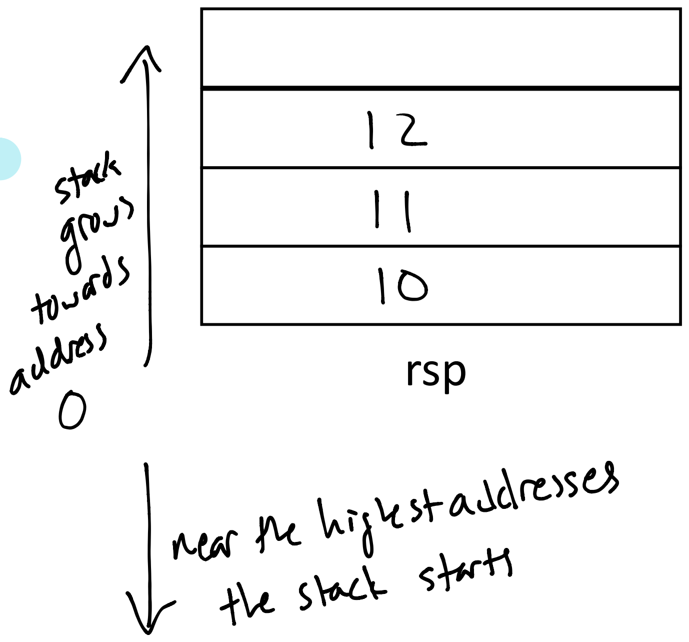
(let (x (read-int)) (inc x))

env is mapping names to values vs. names to locations/addrs

runtime — compile-time

```
(let (x 10)
  (let (y (inc x))
    (let (z (inc y))
      z)))
```

```
mov rax, 10
mov [rsp - 8], rax
mov rax, [rsp - 8]
add rax, 1
mov [rsp - 16], rax
mov rax, [rsp - 16]
add rax, 1
mov [rsp - 24], rax
mov rax, [rsp - 24]
```



stack grows towards address 0
12
11
10
rsp
near the highest addresses the stack starts

```
(let (x (let (y 10) (inc y)))
  (let (z (inc x))
    z))
```

```
(*
expr := <number>
      |  (let (<name> <expr>) <expr>)
      |  (+ <expr> <expr>)
      |  <name>
*)
type expr =
   | ENum of int
   | EId of string
   | ELet of string * expr * expr
   | EPlus of expr * expr
```

```
let stackloc i = (i * 8)
let stackval i = sprintf "[rsp - %d]" (stackloc i)
type tenv = (string * int) list

(* Assume si starts at 1 in the first call *)
let rec e_to_is (e : expr) (si : int) (env : tenv) =
  match e with
    | EPlus(e1, e2) ->
```

```
let e1is = e_to_is e1 si env in
let e2is = e_to_is e2 si env in
e1is @
["mov rbx, rax"] @
e2is @
["add rax, rbx"]
```

```
let e1is = e_to_is e1 si env in
let e2is = e_to_is e2 (si + 1) env in
e1is @
[sprintf "mov %s, rax" (stackval si)] @
e2is @
[sprintf "mov %s, rax" (stackval (si + 1));
 sprintf "mov rax, %s" (stackval si);
 sprintf "add rax, %s" (stackval (si + 1))]
```

(+ 1 2)

```
mov rax, 1
mov rbx, rax
mov rax, 2
add rax, rbx
```

```
let e1is = e_to_is e1 si env in
let e2is = e_to_is e2 si env in
e1is @
["mov rbx, rax"] @
e2is @
["add rax, rbx"]
```

```
let e1is = e_to_is e1 si env in
let e2is = e_to_is e2 (si + 1) env in
e1is @
[sprintf "mov %s, rax" (stackval si)] @
e2is @
[sprintf "mov %s, rax" (stackval (si + 1));
 sprintf "mov rax, %s" (stackval si);
 sprintf "add rax, %s" (stackval (si + 1))]
```

(+ 5 (+ 1 3))

```
mov rax, 5
mov rbx, rax
mov rax, 1
mov rbx, rax
mov rax, 3
add rax, rbx
add rax, rbx
```

Which of these fills in the *parse* case for ELet?

(in sexp_to_expr)

A.
```
| [Atom("let"); Atom(name); e1; e2] ->
 ELet(name, sexp_to_expr e1, sexp_to_expr e2)
```

[✓ B.]
```
| [Atom("let"); List([Atom(name); e1]); e2] ->
 ELet(name, sexp_to_expr e1, sexp_to_expr e2)
```

C.
```
| [Atom("let"); List([Atom(name); e1]); e2] ->
 ELet(EId(name), sexp_to_expr e1, sexp_to_expr e2)
```

D.
```
| [Atom("let"); Atom(name); e1; e2] ->
 ELet(EId(name), sexp_to_expr e1, sexp_to_expr e2)
```

E. None of the above

(let (x 5) (inc x))

defining the name

List([Atom("let"); List([Atom("x"); Atom("5")]);

List([Atom("inc"); Atom("x")])])

using it

EId("x")

- What instructions will we get from running

  expr_to_instrs (EId("y")) 3 [("x", 1); ("y", 2)]

  - A: mov rax, [rsp-8]
  - [✓ B: mov rax, [rsp-16]]
  - C: mov rax, 2
  - D: mov rax, 8
  - E: An error – "Unbound id"

mov rax, [rsp-16]

stack pointer

moves from mem to rax