```
type expr =
  | ENum of int
  | EBool of bool
  | EId of string
  | EIf of expr * expr * expr
  | ELet of string * expr * expr
  | EPlus of expr * expr
  | EApp of string * expr
type def =
  | Def of string * string * expr
type prog =
  | Prog of def list
```

```
(def (double n)
  (+ n n))
(def (g x)
  (let (y (+ x 1))
    (let (z (double y))
      (+ z y))))
(def (our_main input)
  (g input))
```

```
int main(int argc, char** argv) {
  int parsed_input = 0;
  if(argc > 1) { parsed_input = atoi(argv[1]); }
  int result = our_code_starts_here(parsed_input);
  printf("%d\n", result);
  fflush(stdout);
  return 0;
}
```

```
let rec stack_depth e =
```

```
let compile_def (d : def) =
  match d with
    | Def(name, arg, body) ->
      let depth = stack_depth body in
      let bodyis = e_to_is body 2 [(arg, 1)] in
      [
        sprintf "%s:" name;
        sprintf "sub esp, %d" (depth * 4);
      ]
      @ bodyis @
      [
        sprintf "mov esp, ebp";
        "ret"
      ]
let rec e_to_is e si env =
  match e with
    | EApp(name, arg) ->
      let after_label = gen_tmp "after_call" in
      let argis = e_to_is arg si env in
      argis @
      [
        "push ebp";
        sprintf "push %s" after_label;
        "mov ebp, esp";
        "push eax";
        sprintf "jmp %s" name;
        sprintf "%s:" after_label;
        "pop ebp";
      ]
```

g:

```
sub esp, 16
mov eax, [ebp - 4]
mov [ebp - 8], eax
mov eax, 1
mov [ebp - 12], eax
mov eax, [ebp - 8]
add eax, [ebp - 12]
mov [ebp - 8], eax
mov eax, [ebp - 8]
push ebp
push after_call1
mov ebp, esp
push eax
jmp double
```

after_call1:

```
pop ebp
mov [ebp - 12], eax
mov eax, [ebp - 12]
mov [ebp - 16], eax
mov eax, [ebp - 8]
mov [ebp - 20], eax
mov eax, [ebp - 16]
add eax, [ebp - 20]
mov esp, ebp
```

ret

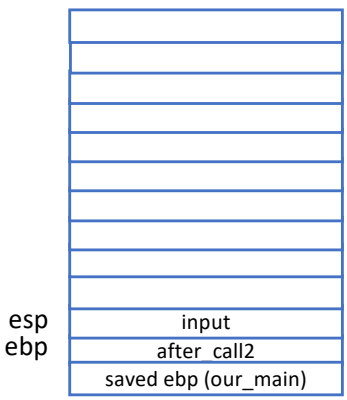
double:

```
sub esp, 8
mov eax, [ebp - 4]
mov [ebp - 8], eax
mov eax, [ebp - 4]
mov [ebp - 12], eax
mov eax, [ebp - 8]
add eax, [ebp - 12]
mov esp, ebp
ret
```

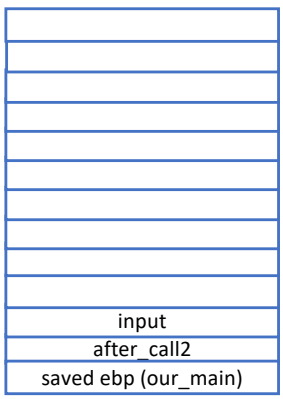
our_main:

```
sub esp, 4
mov eax, [ebp - 4]
push ebp
push after_call2
mov ebp, esp
push eax
jmp g
after_call2:
pop ebp
mov esp, ebp
ret
```

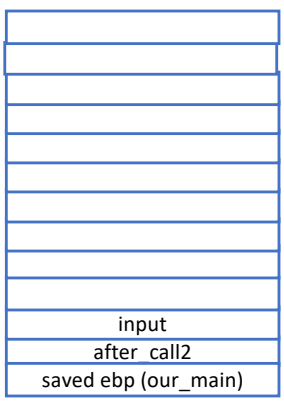
Stack at g:



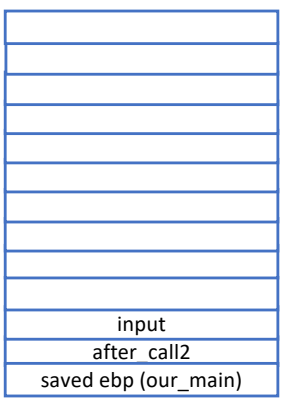
Stack at double:



Stack after return from double (at after_call1)



Stack at ret in g



```
let rec e_to_is e si env =
 (* EApp on front ... *)
| ENum(i) -> [sprintf "mov eax, %d" i]
| EBool(true) -> ["mov eax, 0"]
| EBool(false) -> ["mov eax, 1"]
| EPlus(e1, e2) ->
  let e1is = e_to_is e1 si env in
  let e2is = e_to_is e2 (si + 1) env in
  e1is @
  [sprintf "mov %s, eax" (stackval si)] @
  e2is @
  [sprintf "mov %s, eax" (stackval (si + 1));
   sprintf "mov eax, %s" (stackval si);
   sprintf "add eax, %s" (stackval (si + 1));
   ]
| EId(x) ->
  (match find env x with
    | None -> failwith "Unbound id"
    | Some(i) ->
      [sprintf "mov eax, [ebp - %d]" (stackloc i)])
| ELet(x, v, body) ->
  let vis = e_to_is v si env in
  let bis = e_to_is body (si + 1) ((x,si)::env) in
  vis @
  [sprintf "mov [ebp - %d], eax" (stackloc si)] @
  bis
| EIf(cond, thn, els) ->
  let condis = e_to_is cond si env in
  let afterlabel = gen_tmp "after_if" in
  let elslabel = gen_tmp "else" in
  let thnis = e_to_is thn si env in
  let elsis = e_to_is els si env in
  condis @ [
    "cmp eax, 0";
    sprintf "je %s" elslabel;
  ] @ thnis
  @ [ sprintf "jmp %s" afterlabel; sprintf "%s:" elslabel ]
  @ elsis @ [ sprintf "%s:" afterlabel ]


let compile_prog prog =
  match prog with
    | Prog(defs) ->
      List.concat (List.map compile_def defs)
       (* compile_def is on the front *)

let compile (program : string) : string =
  let ast = parse ("(" ^ program ^ ")") in
  let instrs = compile_prog ast in
  let instrs_str = (String.concat "\n" instrs) in
  sprintf "
section .text
global our_code_starts_here
extern print_error_and_exit
error:
  push eax
  jmp print_error_and_exit
%s
our_code_starts_here:
  mov eax, [esp+4]
  push ebp
  push after_main
  mov ebp, esp
  push eax
  jmp our_main
  after_main:
  pop ebp
  ret\n" instrs_str
```