```
let rec e_to_is e si env (tc: bool) =
  match e with
   | EApp2(name, arg1, arg2) ->
     let after_label = gen_tmp "after_call" in
     let arg1is = e_to_is arg1 si env false in
     let arg2is = e_to_is arg2 (si + 1) env false in
     let init = if tc then
       [ "mov esp, ebp" ]
     else
       [ "push ebp"; sprintf "push %s" after_label; ] in
     let after = if tc then [] else
        [ sprintf "%s:" after_label; "pop ebp"] in
     arg1is @ [ sprintf "mov %s, eax" (stackval si)]  @
     arg2is @ [ sprintf "mov %s, eax" (stackval (si + 1))] @
      init @ [
      "mov ebp, esp";
      sprintf "mov eax, %s" (stackval si); "push eax";
      sprintf "mov eax, %s" (stackval (si + 1)); "push eax";
      sprintf "jmp %s" name;
      ] @
      after
   | EPlus(e1, e2) ->
     let e1is = e_to_is e1 si env false in
     let e2is = e_to_is e2 (si + 1) env false in
     (* code for doing addition *)
   | ELet(x, v, body) ->
     let vis = e_to_is v si env false in
     let bis = e_to_is body (si + 1) ((x,si)::env) tc in
     (* code for saving var, body *)
   | EIf(cond, thn, els) ->
     let condis = e_to_is cond si env false in
     let afterlabel = gen_tmp "after_if" in
     let elslabel = gen_tmp "else" in
     let thnis = e_to_is thn si env tc in
     let elsis = e_to_is els si env tc in
     (* code for checking condition, then, els, etc *)
```

```
let compile_def (d : def) =
  | Def2(name, arg1, arg2, body) ->
    let depth = stack_depth body in
    let env = _____________________ in

    let bodyis = e_to_is body _________ env true in
    [
      sprintf "%s:" name;
      sprintf "sub esp, %d" (depth * 4);
    ]
    @ bodyis @
    [
      sprintf "mov esp, ebp";
      "ret"
    ]

type expr =
  ...
  | EApp2 of string * expr * expr

type def =
  ...
  | Def2 of string * string * string * expr

let rec stack_depth (e : expr) =
  match e with
    | ENum(_) | EBool(_) | EId(_) -> 0
    | ELet(x, v, body) ->
      (max (stack_depth v) ((stack_depth body) + 1))
    | EPlus(lhs, rhs) ->
      (max (stack_depth lhs) ((stack_depth rhs) + 1)) + 1
    | EApp2(name, arg1, arg2) ->
      ____________________________________________
    | EIf(cond, thn, els) ->
      (max (max (stack_depth cond) (stack_depth thn))
           (stack_depth els))
```

```
(def (sum n _)
 (if n
  (+ n (sum (+ n -1) _))
  0))

(def (our_main input)
 (sum 3 0))
```

(Note – the "_" argument is just to match the EApp2 case, so we just have to look at one case for applications)

```
sum:
sub esp, 20
mov eax, [ebp - 4]
cmp eax, 0
je else2
mov eax, [ebp - 4]
mov [ebp - 8], eax
mov eax, [ebp - 4]
mov [ebp - 12], eax
mov eax, -1
mov [ebp - 16], eax
mov eax, [ebp - 12]
add eax, [ebp - 16]
```

```
_____________________________
mov ebp, esp
push eax
jmp sum
```

```
_____________________________
mov [ebp - 12], eax
mov eax, [ebp - 8]
add eax, [ebp - 12]
jmp after_if1
else2:
mov eax, 0
after_if1:
mov esp, ebp
ret
```

```
(def (sum n sofar)
 (if n
  (sum (+ n -1) (+ n sofar))
  sofar))

(def (our_main input)
 (sum 3 0))
```

```
sum:
sub esp, 12
mov eax, [ebp - 4]
cmp eax, 0
je else2
mov eax, [ebp - 4]
mov [ebp - 12], eax
mov eax, -1
mov [ebp - 16], eax
mov eax, [ebp - 12]
add eax, [ebp - 16]
mov [ebp - 12], eax
mov eax, [ebp - 4]
mov [ebp - 16], eax
mov eax, [ebp - 8]
mov [ebp - 20], eax
mov eax, [ebp - 16]
add eax, [ebp - 20]
mov [ebp - 16], eax
```

```
_____________________________
mov ebp, esp
mov eax, [ebp - 12]
push eax
mov eax, [ebp - 16]
push eax
jmp sum
```

```
_____________________________
jmp after_if1
else2:
mov eax, [ebp - 8]
after_if1:
mov esp, ebp
ret
```

```
(def (sum n _)
  (if n
    (+ n (sum (+ n -1) _))
    0))

(def (our_main input)
  (sum 3 0))
```

(Note – the "_" argument is just to match the EApp2 case, so we just have to look at one case for applications)

```
sum:
sub esp, 20
mov eax, [ebp - 4]
cmp eax, 0
je else2
mov eax, [ebp - 4]
mov [ebp - 8], eax
mov eax, [ebp - 4]
mov [ebp - 12], eax
mov eax, -1
mov [ebp - 16], eax
mov eax, [ebp - 12]
add eax, [ebp - 16]
```

```
_____________________________
mov ebp, esp
push eax
jmp sum
```

```
_____________________________
mov [ebp - 12], eax
mov eax, [ebp - 8]
add eax, [ebp - 12]
jmp after_if1
else2:
mov eax, 0
after_if1:
mov esp, ebp
ret
```

```
(def (sum n sofar)
  (if n
    (sum (+ n -1) (+ n sofar))
    sofar))

(def (our_main input)
  (sum 3 0))
```

```
sum:
sub esp, 12
mov eax, [ebp - 4]
cmp eax, 0
je else2
mov eax, [ebp - 4]
mov [ebp - 12], eax
mov eax, -1
mov [ebp - 16], eax
mov eax, [ebp - 12]
add eax, [ebp - 16]
mov [ebp - 12], eax
mov eax, [ebp - 4]
mov [ebp - 16], eax
mov eax, [ebp - 8]
mov [ebp - 20], eax
mov eax, [ebp - 16]
add eax, [ebp - 20]
mov [ebp - 16], eax
```

```
_____________________________
mov ebp, esp
mov eax, [ebp - 12]
push eax
mov eax, [ebp - 16]
push eax
jmp sum
```

```
_____________________________
jmp after_if1
else2:
mov eax, [ebp - 8]
after_if1:
mov esp, ebp
ret
```

```
let rec e_to_is e si env (tc: bool) =
  match e with
    | EApp2(name, arg1, arg2) ->
      let after_label = gen_tmp "after_call" in
      let arg1is = e_to_is arg1 si env false in
      let arg2is = e_to_is arg2 (si + 1) env false in
      let init = if tc then
        [ "mov esp, ebp" ]
      else
        [ "push ebp"; sprintf "push %s" after_label; ] in
      let after = if tc then [] else
         [ sprintf "%s:" after_label; "pop ebp"] in
      arg1is @ [ sprintf "mov %s, eax" (stackval si)]  @
      arg2is @ [ sprintf "mov %s, eax" (stackval (si + 1))] @
       init @ [
       "mov ebp, esp";
       sprintf "mov eax, %s" (stackval si); "push eax";
       sprintf "mov eax, %s" (stackval (si + 1)); "push eax";
       sprintf "jmp %s" name;
       ] @
       after
    | EPlus(e1, e2) ->
      let e1is = e_to_is e1 si env false in
      let e2is = e_to_is e2 (si + 1) env false in
      (* code for doing addition *)
    | ELet(x, v, body) ->
      let vis = e_to_is v si env false in
      let bis = e_to_is body (si + 1) ((x,si)::env) tc in
      (* code for saving var, body *)
    | EIf(cond, thn, els) ->
      let condis = e_to_is cond si env false in
      let afterlabel = gen_tmp "after_if" in
      let elslabel = gen_tmp "else" in
      let thnis = e_to_is thn si env tc in
      let elsis = e_to_is els si env tc in
      (* code for checking condition, then, els, etc *)
```

```
let compile_def (d : def) =
  | Def2(name, arg1, arg2, body) ->
    let depth = stack_depth body in
    let env = _____________________ in

    let bodyis = e_to_is body _________ env true in
    [
      sprintf "%s:" name;
      sprintf "sub esp, %d" (depth * 4);
    ]
    @ bodyis @
    [
      sprintf "mov esp, ebp";
      "ret"
    ]

type expr =
  ...
  | EApp2 of string * expr * expr

type def =
  ...
  | Def2 of string * string * string * expr

let rec stack_depth (e : expr) =
  match e with
    | ENum(_) | EBool(_) | EId(_) -> 0
    | ELet(x, v, body) ->
      (max (stack_depth v) ((stack_depth body) + 1))
    | EPlus(lhs, rhs) ->
      (max (stack_depth lhs) ((stack_depth rhs) + 1)) + 1
    | EApp2(name, arg1, arg2) ->
      ____________________________________________
    | EIf(cond, thn, els) ->
      (max (max (stack_depth cond) (stack_depth thn))
           (stack_depth els))
```