

Mid-May  
(with some changes)

↓

```
type expr =  
  | ENum of int  
  | EBool of bool  
  | EDef of string * string * expr  
  | EApp of string * expr
```

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

```
def f(x):  
  g(x + 2) + 3
```

```
f(y + 4)
```

fun name to call    argument value

fun name defined    function body

argument name

This week

↙

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

Which representation do you want to implement first?

Some things to discuss:

- Compiling the new abstract syntax (getting its answer into EAX)
- How the environment works (a new kind of name)
- Are there programs we can represent with one but not the other?

```
type expr =  
  | ENum of int  
  | EBool of bool  
  ...  
  | EApp of string * expr
```

```
type def =  
  | Def of string * string * expr
```

```
type prog =  
  | Prog of def list * expr
```

```
let rec compile_expr e si env =  
  match e with  
    ...  
    | EApp(fname, arg) -> ...
```

```
let compile_def d ... =  
  ...
```

```
let compile_prog p ... =  
  ...
```

```
let rec compile_expr e si env =  
  match e with  
    ...  
    | EApp(fname, arg) ->  
      INSTRUCTIONS FOR FUNCTION CALL
```

```
let compile_def d ... =  
  INSTRUCTIONS FOR FUNCTION BODY
```

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

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

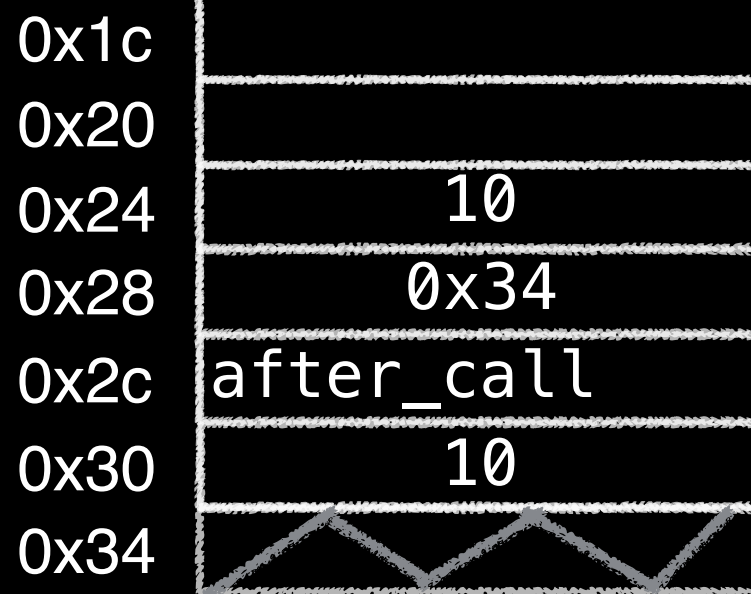
```
mov eax, 10
mov [esp-4], eax
mov eax, [esp-4]
mov [esp-8], after_call
mov [esp-12], esp
mov [esp-16], eax
sub esp, 8
jmp g
after_call:
```

esp ~~0x34~~ 0x2c

eax ~~10~~ 10

g:

Where is the  
argument y  
in terms of the  
**current value**  
of esp?



- A: esp
- B: esp-4
- C: esp-8
- D: esp+4
- E: esp-12

```
let rec compile_expr e si env =
  match e with
```

```
let compile_def d ... =
  INSTRUCTIONS FOR FUNCTION BODY
```

```
...
| EApp(fname, arg) ->
  INSTRUCTIONS FOR FUNCTION CALL
```

```

let x = 10 in
let z = g(x) in
3 + z

```

```

def g(y):
    y + 1

```

```

mov eax, 10
mov [esp-4], eax
mov eax, [esp-4]
mov [esp-8], after_call
mov [esp-12], esp
mov [esp-16], eax
sub esp, 8
jmp g
after_call:
mov esp, [esp-8]
mov [esp-8], eax
mov eax, 3
add eax, [esp-8]

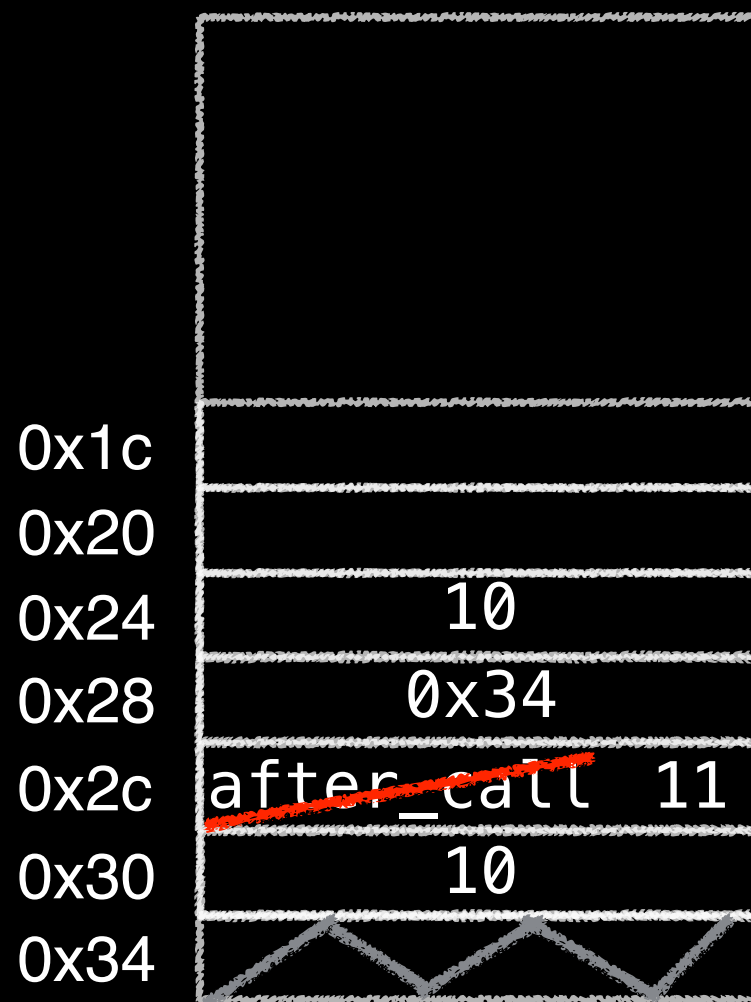
```

esp	<del>0x34</del>	<del>0x2c</del>	<del>0x30</del>	<del>0x34</del>
eax	<del>10</del>	<del>10</del>	<del>11</del>	<del>3</del> 14

```

g:
mov eax, [esp-8]
add eax, 1
ret

```



Where is the  
**current value**  
of [esp-8]?

- A: 10
- B: 0x34
- C: after\_call
- D: the other 10

```

let rec compile_expr e si env =
  match e with

```

```

let compile_def d ... =
  INSTRUCTIONS FOR FUNCTION BODY

```

```

...
| EApp(fname, arg) ->
  INSTRUCTIONS FOR FUNCTION CALL

```

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

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

```
mov eax, 10
mov [esp-4], eax
```

```
mov eax, [esp-4]
```

```
mov [esp-8], after_call
```

```
mov [esp-12], esp
```

```
mov [esp-16], eax
```

```
sub esp, 8
```

```
jmp g
```

```
after_call:
```

```
mov esp, [esp-8]
```

```
mov [esp-8], eax
```

```
mov eax, 3
```

```
add eax, [esp-8]
```

```
esp 0x34 0x2c 0x30 0x34
```

```
eax 10 10 11 3 14
```

```
g:
mov eax, [esp-8]
```

## Call setup:

- Always these 3 values
- Always this order
- Always start at current si
- Always subtract to point esp at the return address

0x24	10
0x28	0x34
0x2c	<del>after_call</del> 11
0x30	10
0x34	

```
let rec compile_expr e si env =
  match e with
```

```
let compile_def d ... =
  INSTRUCTIONS FOR FUNCTION BODY
```

```
...
| EApp(fname, arg) ->
  INSTRUCTIONS FOR FUNCTION CALL
```



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

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

```
mov eax, 10
mov [esp-4], eax
mov eax, [esp-4]
mov [esp-8], after_call
mov [esp-12], 11
mov [esp-16], 10
sub esp, 8
jmp g
after_call
mov esp, [esp-4]
```

esp	<del>0x34</del>	<del>0x2c</del>	<del>0x30</del>	<del>0x34</del>
eax	<del>10</del>	<del>10</del>	<del>11</del>	<del>3</del> 14

```
g:
mov eax, [esp-8]
add eax, 1
ret
```

Callee has an easy job:

- Rely on (first) argument in [esp-8], so env starts with [(arg, 2)]
- Start at a “higher” si=3 for any local vars
- Expect [esp] to contain return pointer, use ret

mov [esp-8], eax	0x2c	<del>after_call</del> 11
mov eax, 3	0x30	10
add eax, [esp-8]	0x34	

```
let rec compile_expr e si env =
  match e with
```

```
let compile_def d ... =
  INSTRUCTIONS FOR FUNCTION BODY
```

```
...
| EApp(fname, arg) ->
  INSTRUCTIONS FOR FUNCTION CALL
```

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

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

```
mov eax, 10
mov [esp-4], eax
mov eax, [esp-4]
mov [esp-8], after_call
mov [esp-12], 11
mov [esp-16], 3
sub esp, 8
jmp g
```

esp	<del>0x34</del>	<del>0x2c</del>	<del>0x30</del>	0x34
eax	<del>10</del>	<del>10</del>	<del>11</del>	<del>3</del> 14

```
g:
mov eax, [esp-8]
add eax, 1
ret
```

After the call:

- Rely on old esp at [esp-8] (always)
- Expect answer to be in eax from callee

```
after_call:
mov esp, [esp-8]
```

```
mov [esp-8], eax
mov eax, 3
add eax, [esp-8]
```

0x20	
0x24	10
0x28	0x34
0x2c	<del>after_call</del> 11
0x30	10
0x34	

```
let rec compile_expr e si env =
  match e with
```

```
let compile_def d ... =
  INSTRUCTIONS FOR FUNCTION BODY
```

```
...
| EApp(fname, arg) ->
  INSTRUCTIONS FOR FUNCTION CALL
```

```

let x = 10 in
let z = g(x) in
3 + z

```

```

def g(y):
    y + 1

```

```

mov eax, 10
mov [esp-4], eax
    mov eax, [esp-4]
    mov [esp-8], after_call
    mov [esp-12], esp
    mov [esp-16], eax
    sub esp, 8
    jmp g
after_call:
    mov esp, [esp-8]
mov [esp-8], eax
mov eax, 3
add eax, [esp-8]

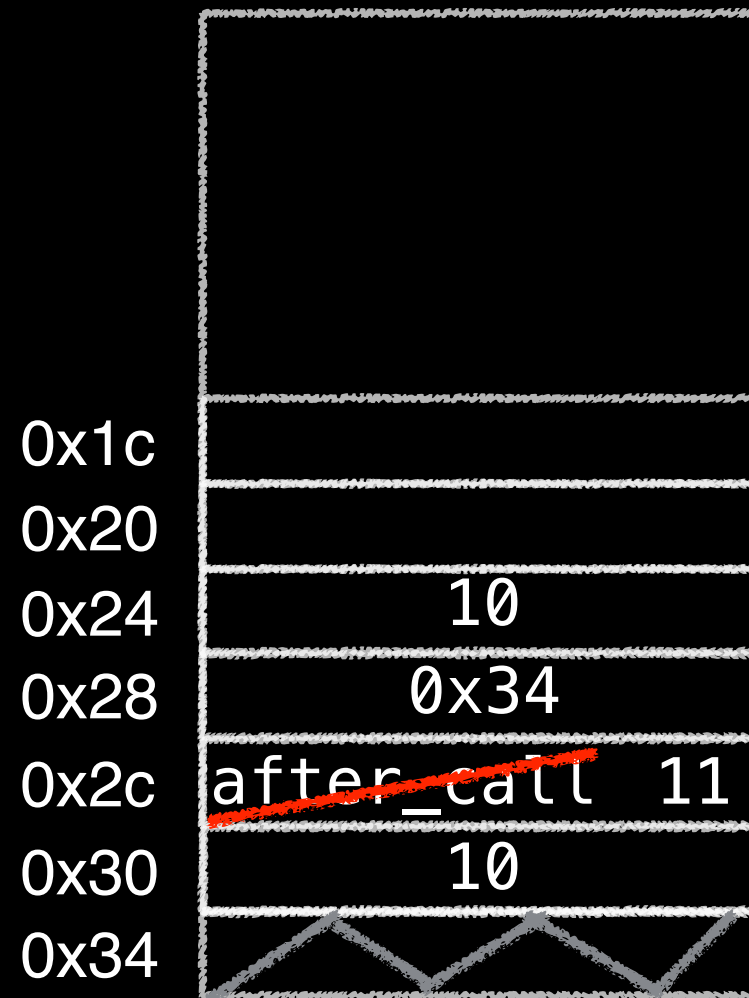
```

esp	<del>0x34</del>	<del>0x2c</del>	<del>0x30</del>	0x34
eax	<del>10</del>	<del>10</del>	<del>11</del>	<del>3</del> 14

```

g:
mov eax, [esp-8]
add eax, 1
ret

```



```

let rec compile_expr e si env =
  match e with

```

```

    let compile_def d ... =
      INSTRUCTIONS FOR FUNCTION BODY

```

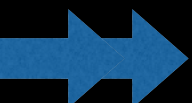
```

    ...
    | EApp(fname, arg) ->
      INSTRUCTIONS FOR FUNCTION CALL

```

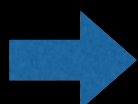






sum:

```
mov eax, [esp-8]
cmp eax, 1
jne false_branch
mov eax, 1
jmp after_if
false_branch:
mov eax, [esp-8]
sub eax, 1
mov [esp-12], after_call
mov [esp-16], esp
mov [esp-20], eax
sub eax, 12
jmp sum
after_call:
mov esp, [esp-8]
mov [esp-12], eax
mov eax, [esp-8]
add eax, [esp-12]
after_if:
ret
```



def sum(x):

```
if x == 1: 1
else:
    x + sum(x - 1)
```

eq

~~NO~~ ~~NO~~ YES

esp

~~0x30~~ ~~0x24~~ ~~0x18~~ ~~0x1c~~  
~~0x24~~ ~~0x28~~ ~~0x30~~ 0x34

eax

~~3~~ ~~3~~ ~~2~~ ~~1~~ ~~2~~ ~~3~~ ~~3~~ 6

0x10

1

0x14

0x24

0x18

~~after\_call~~ 1

0x1c

2

0x20

0x30

0x24

~~after\_call~~ 3

0x28

3

0x2c

old\_esp

0x30

return\_ptr

0x34