

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

fun name to call | argument value

```
def f(x):  
    g(x + 2) + 3  
  
f(y + 4)
```

fun name defined | function body

argument name

Mid-May
(with some changes)

↓

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

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
```

```
mov eax, 10
```

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

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

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

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

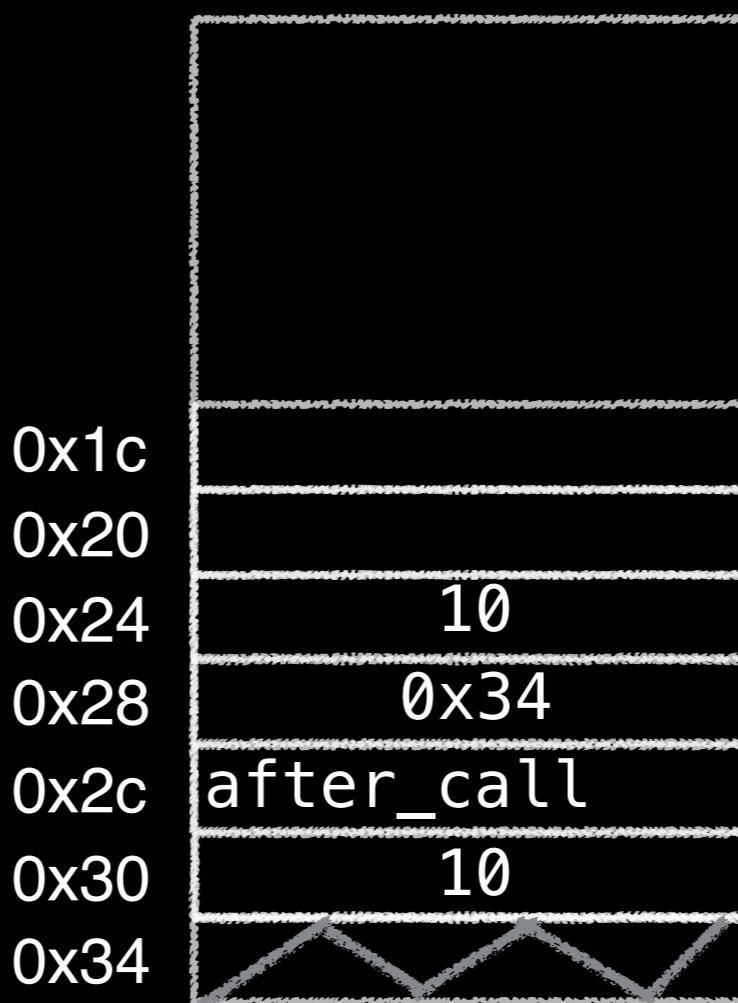
```
mov [esp-16], eax
```

```
sub esp, 8
```

```
jmp g
```

```
after_call:
```

esp	0x34 0x2c
eax	10 10



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

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
```

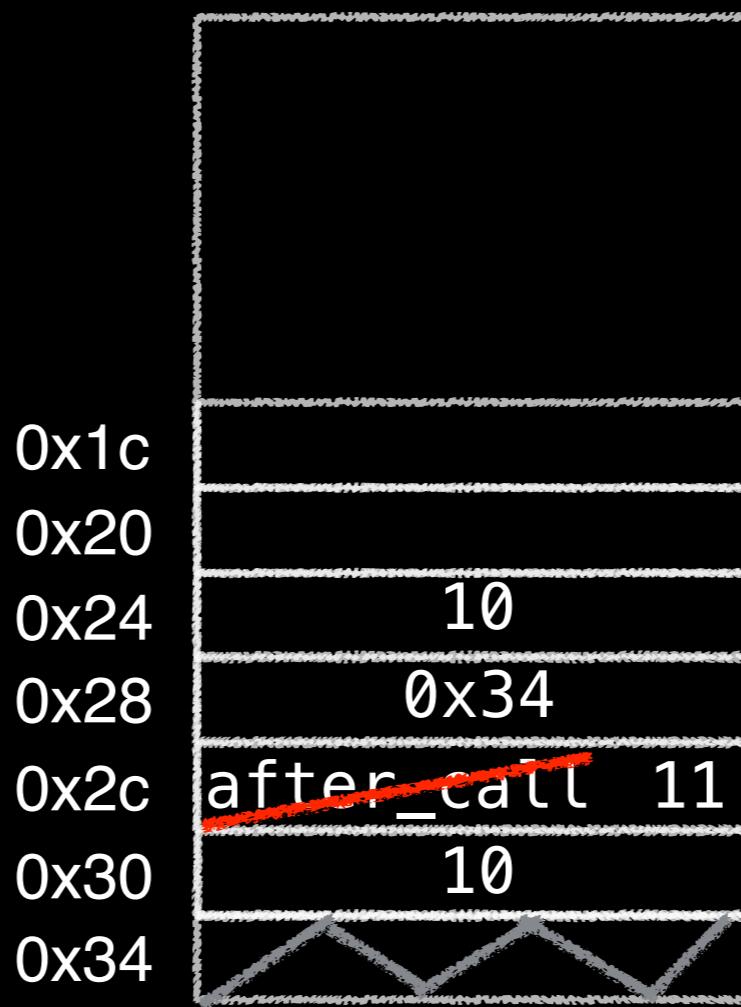
...
| EApp(fname, arg) ->
INSTRUCTIONS FOR FUNCTION CALL

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

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

```
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



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

```
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
```

...
| 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
```

```
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

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

0x14	10
0x24	10
0x28	0x34
0x2c	after_call 11
0x30	10
0x34	

```
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
```

```
mov eax, 10  
mov [esp-4], eax  
mov eax, [esp-4]  
mov [esp-8], after_call
```

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

0x2c	after_call 11
0x30	10
0x34	

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

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

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

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

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

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

```
mov eax, 10  
mov [esp-4], eax  
mov eax, [esp-4]  
mov [esp-8], after_call
```

After the call:

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

```
jmp g
```

```
after_call:
```

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

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

```
    mov eax, 3
```

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

esp 

eax 

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

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

[]

0x20	
0x24	10
0x28	0x34
0x2c	after_call 11
0x30	10
0x34	

```
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
```

```
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

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

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

The diagram shows a vertical stack of memory cells. The addresses listed on the left are 0x1c, 0x20, 0x24, 0x28, 0x2c, 0x30, and 0x34. The values in the cells are: 0x1c (empty), 0x20 (empty), 0x24 (10), 0x28 (0x34), 0x2c (after_call 11, with 'after_call' crossed out), 0x30 (10), and 0x34 (empty). A red box highlights the first four cells (0x34, 0x2c, 0x30, 0x34).

0x1c	
0x20	
0x24	10
0x28	0x34
0x2c	after_call 11
0x30	10
0x34	

```
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
```



```
def twosqrt3x(x):
    let ans = do_sqrt(x * 3) in
        2 * ans
twosqrt3x(4)
```

```
twosqrt3x:
    mov eax, [esp-8]
    imul eax, 3
    sub esp, 8
    push eax
    call do_sqrt
```

0x900

```
mov eax, 4
mov [esp-4], after_call
mov [esp-8], esp
mov [esp-12], eax
sub esp, 4
jmp twosqrt3x
after_call:
```

```
int do_sqrt(int val) {
    float asF = (float)val;
    return (int)(sqrt(asF));
}
```

esp ~~0x34 0x30 0x28 0x24~~
eax ~~+ 12~~

0x10	
0x14	
0x18	
0x1c	
0x20	0x900 (ret address)
0x24	12
0x28	4
0x2c	0x34
0x30	after_call
0x34	

```
def twosqrt3x(x):
    let ans = do_sqrt(x * 3) in
    2 * ans
twosqrt3x(4)
```

```
twosqrt3x:
    mov eax, [esp-8]
    imul eax, 3
    sub esp, 8
    push eax
    call do_sqrt
```

0x900

```
int do_sqrt(int val) {
    float asF = (float)val;
    return (int)(sqrt(asF));
}
```

esp C Happens
eax C Happens

What will be in ESP when do_sqrt is complete?

- A: 0x24 (the value of ESP before call)
- B: 0x20 (do_sqrt's return address)
- C: 0x30 (the original ret address)
- D: We can't possibly know
- E: None of the above

after_call:

0x10
0x14
0x18
0x1c
0x20 C Happens 0x900 (ret address)
0x24 12
0x28 4
0x2c 0x34
0x30 after_call
0x34

```
def twosqrt3x(x):
    let ans = do_sqrt(x * 3) in
    2 * ans
twosqrt3x(4)
```

```
twosqrt3x:
    mov eax, [esp-8]
    imul eax, 3
    sub esp, 8
    push eax
    call do_sqrt
```

0x900

```
int do_sqrt(int val) {
    float asF = (float)val;
    return (int)(sqrt(asF));
}
```

esp C Happens
eax C Happens

What will be in EAX when do_sqrt is complete?

- A: We can't possibly know
- B: 3 (the return value)
- C: 12
- D: 4
- E: None of the above

after_call:

0x10
0x14
0x18
0x1c
0x20 C Happens
0x24 0x900 (ret address)
0x28 12
0x2c 4
0x30 0x34
0x34 after_call

```
def twosqrt3x(x):
    let ans = do_sqrt(x * 3) in
    2 * ans
twosqrt3x(4)
```

```
twosqrt3x:
    mov eax, [esp-8]
    imul eax, 3
    sub esp, 8
    push eax
    call do_sqrt
```

0x900

Just as it was
before call

```
mov eax, 4
mov [esp-4], after_call
mov [esp-8], esp
mov [esp-12], eax
sub esp, 4
jmp twosqrt3x
after_call:
```

```
int do_sqrt(int val) {
    float asF = (float)val;
    return (int)(sqrt(asF));
}
```

esp

0x34 0x30 0x28 0x24

eax

+ 12 3

0x10
0x14
0x18
0x1c
0x20
0x24
0x28
0x2c
0x30
0x34

0x900 (ret address)
12
4
0x34
after_call

Return
value

```

def twosqrt3x(x):
    let ans = do_sqrt(x * 3) in
        2 * ans
twosqrt3x(4)

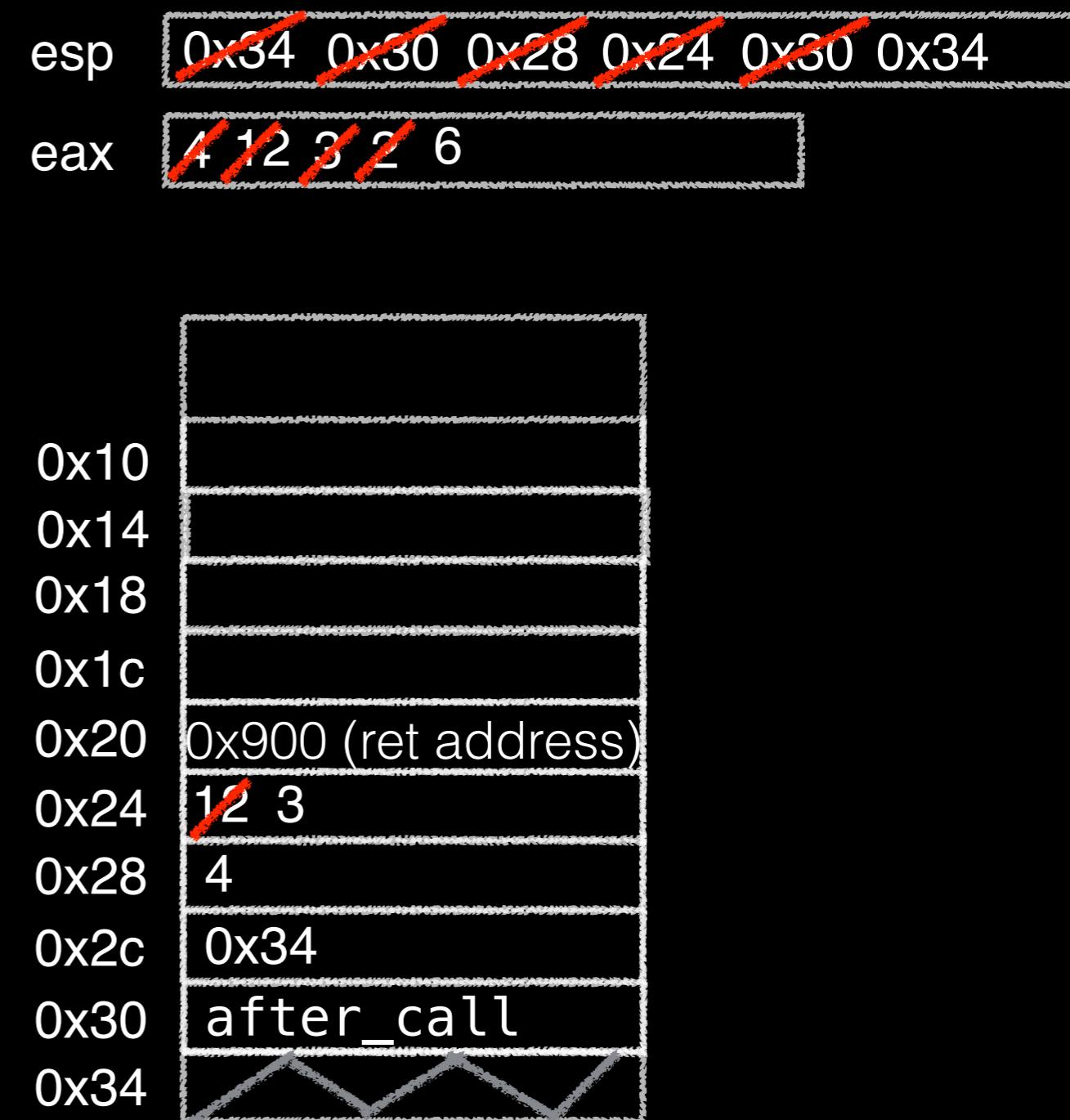
twosqrt3x:
    mov eax, [esp-8]
    imul eax, 3
    sub esp, 8
    push eax
    call do_sqrt
0x900    add esp, 12
    mov [esp-12], eax
    mov eax, 2
    imul eax, [esp-12]
    ret
    mov eax, 4
    mov [esp-4], after_call
    mov [esp-8], esp
    mov [esp-12], eax
    sub esp, 4
    jmp twosqrt3x
after_call:
    mov esp, [esp-8]
    ret

```

```

int do_sqrt(int val) {
    float asF = (float)val;
    return (int)(sqrt(asF));
}

```



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 esp, 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 6

What values will be stored in 0x10, 0x14, 0x18 by the next three instructions?

- A: 1, 0x20, after_call
- B: 2, 0x24, after_call
- C: 2, 0x20, after_call
- D: 1, 0x24, after_call

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


```

def twosqrt3x(x):
    let ans = do_sqrt(x * 3) in
    2 * ans
twosqrt3x(4)
twosqrt3x:
    mov eax, [esp-8]
    imul eax, 3
    sub esp, 8
    push eax
    call do_sqrt
0x900    add esp, 12
    mov [esp-12], eax
    mov eax, 2
    imul eax, [esp-12]
    ret
    mov eax, 4
    mov [esp-4], after_call
    mov [esp-8], esp
    mov [esp-12], eax
    sub esp, 4
    jmp twosqrt3x
after_call:
    mov esp, [esp-8]
    ret

```

Inter-EBP
Range

```

int do_sqrt(int val) {
    float asD = (float)val;
    return (int)(sqrt(asD));
}

```

