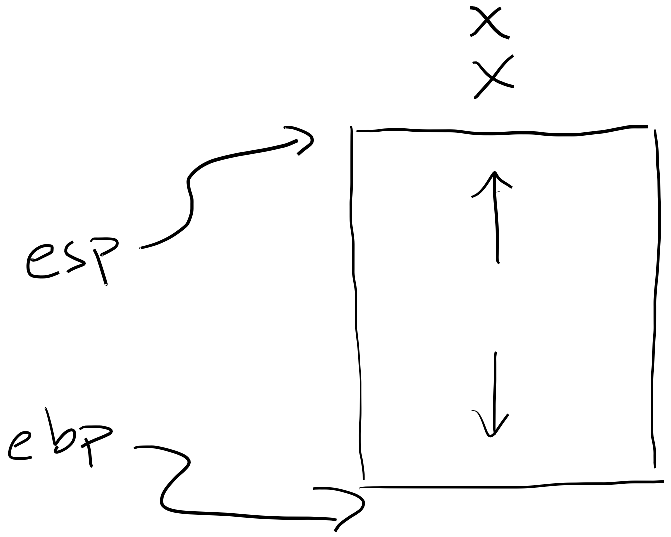
```
mov [esp-4], eax
```



Which of these is NOT true of this compiler?

~~esp~~

mov [ebp-4], eax

A: This compiler uses EBP instead of ESP as the base for variable lookups

sub esp, 20

B: The first instruction of each function subtracts from ESP to make room for local variables

our_main:
  sub esp, 4

~~C~~: Function definitions allow for more than one argument in the abstract syntax

Def of string * string * expr

fun name

arg name

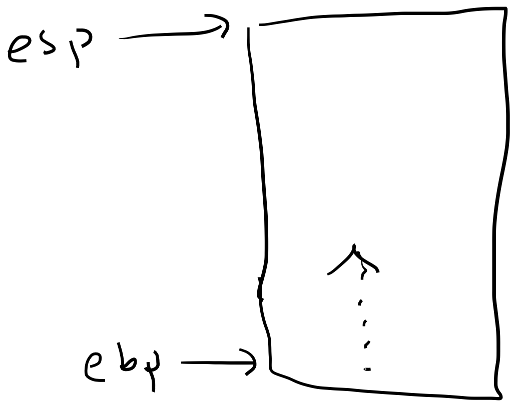
body

~~D~~: There is no way to define recursive functions in this compiler

E: In an application (EApp), the function can only be provided as a name, not an expression

EApp of string * expr

name of called function



else if

```
(if cond
    thn
    (if cond2
        then2
        else2))
```

(or what ~~should~~ should)

What will be the result for running the program on the worksheet with input = 4?

A: 11

→ B: 0

[✓ C: 10]

D: 9

E: 24

```
(def (f x)
   (if x
      (+ x (f (+ x -1)))
      0))
(def (our_main input)
   (f input))
```

(def (f x) ⟺ Def("f", "x"

if takes false branch on 0, then otherwise

Which would fill in the four blanks in the assembly left-hand column? (Look at the EApp case)

A

```
push ebp
push after_call3
mov ebp, esp
push eax
```

B

```
push eax
push ebp
push after_call3
mov ebp, esp
```

C

```
mov ebp, esp
push eax
push ebp
push after_call3
```

D

```
mov ebp, esp
push ebp
push after_call3
push eax
```

```
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";
      ]
```

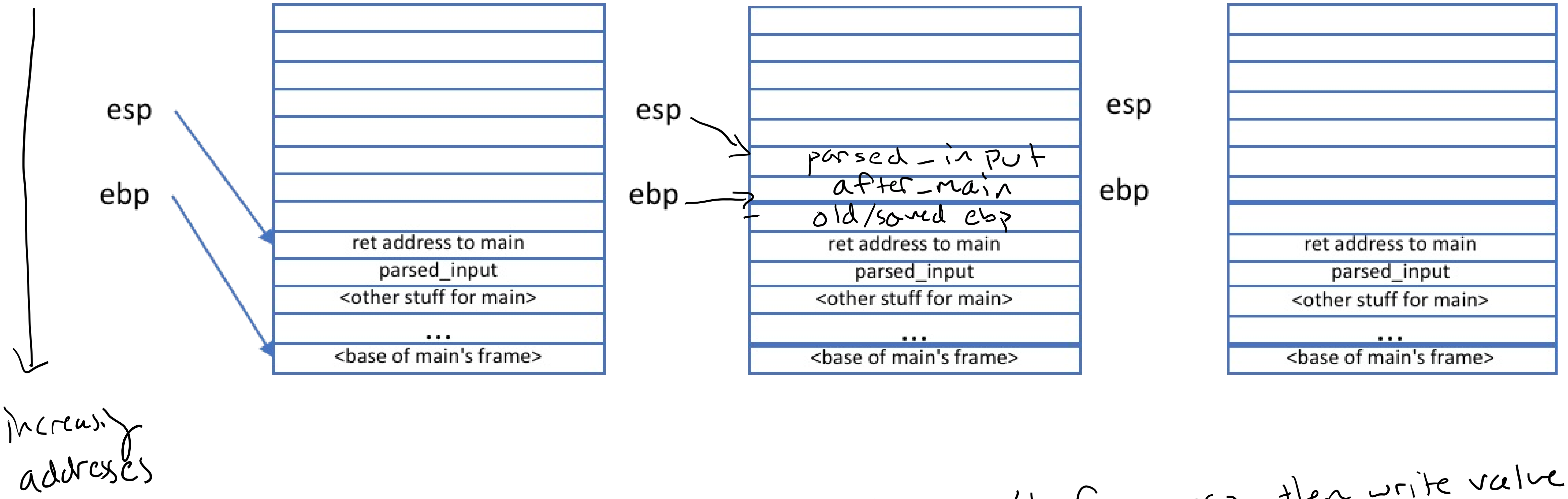
f:

```
sub esp, 20
mov eax, [ebp - 4]
cmp eax, 0              if comp
je else2
mov eax, [ebp - 4]
mov [ebp - 8], eax
mov eax, [ebp - 4]
mov [ebp - 12], eax
mov eax, -1                          calculating
mov [ebp - 16], eax                  argument to f
mov eax, [ebp - 12]
add eax, [ebp - 16]
____push ebp________
____push after-call3_____
____mov ebp, esp____
____push eax________
jmp f
after_call3:                then
pop ebp                     case
mov [ebp - 12], eax
mov eax, [ebp - 8]
add eax, [ebp - 12]
jmp after_if1
else2:                      else
mov eax, 0
after_if1:
mov esp, ebp
ret
```

Stack at our_code_starts_here

Stack at our_main (you fill)

Stack at f (first time, you fill)



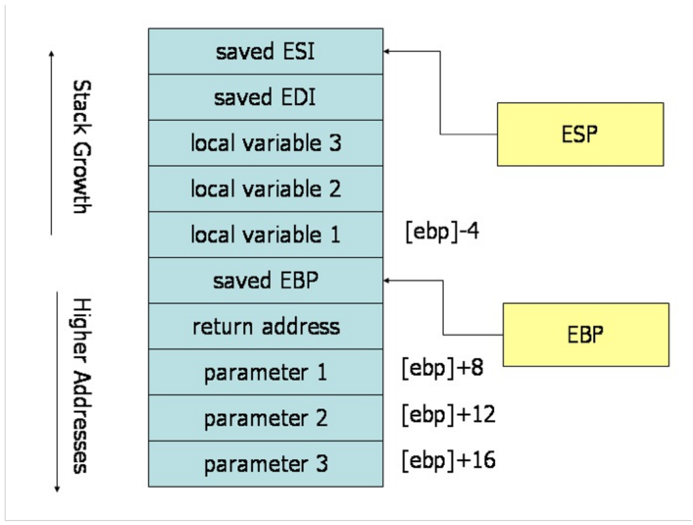
A parsed_input

B current ebp value

push - subtract 4 from esp, then write value into [esp]

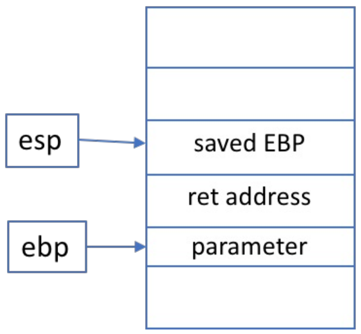
pop - get value and store, then add 4 to esp

C calling on x86

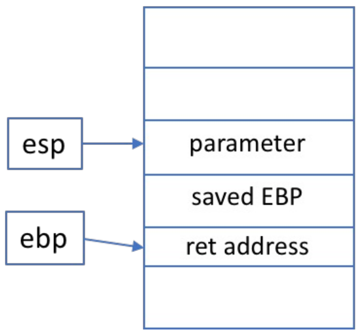


Which represents the calling convention on the worksheet, at the point of jumping to a function?

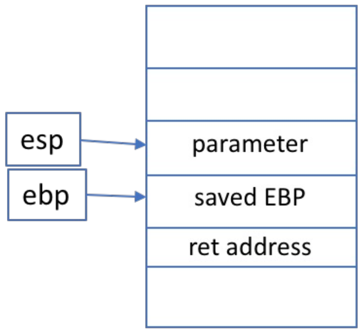
A



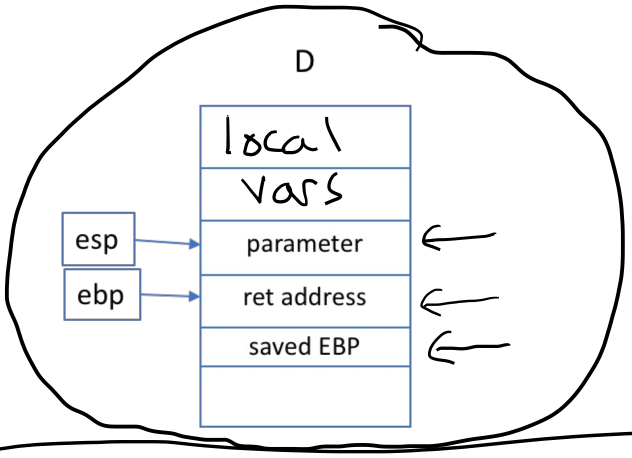
B



C



D



At what address do we find the first (only) parameter?

A: [ESP – 4]
B: [EBP – 4]
C: [ESP + 4]
D: [EBP + 4]
E: something else

In this calling convention, which side, caller or callee, is responsible for saving and resetting EBP to its value before the call?

A: Caller
B: Callee

In this calling convention, which side, caller or callee, is responsible for moving ESP back "below" the local variables and arguments?

A: Caller
B: Callee

In this calling convention, what is the address of the first local variable in a function?

A: [ESP – 4]
B: [EBP – 4]
C: [ESP - 8]
D: [EBP - 12]
E: something else