RUNTIME ORGANIZATION (Solutions)

In the Objective Caml language, we can define the following functions.

```
(* Compute n choose k *)
let binom n k = ...

let test x y =
  let a = binom x y in
  let b = binom x (y+1) in
    a + b  (* Return the sum *)
```

If we compile this code and then disassemble the result we get the following (using Intel syntax):

```
08049950 <camlTest__test_61>:
 8049950:       sub    esp,0xc                        ; esp -= 12
 8049953:       mov    DWORD PTR [esp+0x4],eax        ; *(esp+4) = eax
 8049957:       mov    DWORD PTR [esp],ebx            ; *esp = ebx
 804995a:       call   8049930 <camlTest__binom_58>   ; call binom
 804995f:       mov    DWORD PTR [esp+0x8],eax        ; *(esp+8) = eax
 8049963:       mov    ebx,DWORD PTR [esp]            ; ebx = *esp
 8049966:       add    ebx,0x2                        ; ebx += 2
 8049969:       mov    eax,DWORD PTR [esp+0x4]
 804996d:       call   8049930 <camlTest__binom_58>
 8049972:       mov    ebx,DWORD PTR [esp+0x8]
 8049976:       lea    eax,[ebx+eax*1-0x1]            ; eax = ebx + eax - 1
 804997a:       add    esp,0xc
 804997d:       ret
```

Describe the calling conventions used for `binom`. Where are the arguments `n` and `k` stored? Where is the result located on return?

> *The argument* `n` *is passed in the eax register, and* `k` *in ebx. We can tell this because eax has the same value both times* `binom` *is called, but the second time ebx's value is incremented (by 2 instead of 1 because OCaml uses the least significant bit as a tag bit). The return value is passed in eax, which we can tell because we can trace the two values added for* `a + b` *back to the values of eax right after the two calls to* `binom`*.*

Draw a diagram showing the layout of the stack and the register file right before the second call to `binom`. Your diagram should show where each argument and local variable is stored.

Stack (growing downward):

RA
a
x
y

Registers:

eax: `x`
ebx: `y + 1`

Now consider the following assembly code.

```
080483b4 <test>:
 80483b4:       push   ebp                          ; esp -= 4; *esp = ebp
 80483b5:       mov    ebp,esp
 80483b7:       sub    esp,0x18
 80483ba:       mov    DWORD PTR [ebp-0x14],ecx
 80483bd:       mov    DWORD PTR [ebp-0x18],edx
 80483c0:       mov    eax,DWORD PTR [ebp-0x14]
 80483c3:       mov    edx,DWORD PTR [eax]
 80483c5:       mov    eax,DWORD PTR [ebp-0x18]
 80483c8:       mov    eax,DWORD PTR [eax]
 80483ca:       imul   edx,eax                      ; edx = edx * eax
 80483cd:       mov    eax,DWORD PTR [ebp-0x14]
 80483d0:       mov    ecx,DWORD PTR [eax+0x4]
 80483d3:       mov    eax,DWORD PTR [ebp-0x18]
 80483d6:       mov    eax,DWORD PTR [eax+0x4]
 80483d9:       imul   eax,ecx
 80483dc:       lea    eax,[edx+eax*1]              ; eax = edx + eax
 80483df:       mov    DWORD PTR [ebp-0x4],eax
 80483e2:       mov    eax,DWORD PTR [ebp-0x4]
 80483e5:       leave                               ; esp = ebp; pop ebp
 80483e6:       ret
```

This function takes two parameters, passed in registers `ecx` and `edx` respectively. Its result is returned in register `eax`. Decompile (translate) this assembly into equivalent C code. Hint: This code implements a well-known mathematical operation.

The function implements a two-dimensional dot product.

```
struct vector_t {
  int x;
  int y;
};

int dot(struct vector_t* u, struct vector_t* v) {
  int p;
  p = u->x * v->x + u->y * v->y;
  return p;
}
```

Reasoning: The function uses three different offsets from ebp, so we have three local variables. Two of these (ebp-0x14 and ebp-0x18) correspond to the arguments; we know this because the argument registers are stored to them and they are never overwritten. The third variable (ebp-0x4) is an int because it stores a sum of products of 32-bit integers.

We can tell the arguments are pointers because we dereference them. We use two offsets with each of them and treat the dereferenced values as ints, so we know they point to int arrays or structs with int members. We interpret them as structs here because we recognize the form of the dot product, but it would be equally correct to give the arguments type `int *`*.*