# Part B: Evaluation (20 points)

In this section, write all answers inside OCaml comments. (If you don't, your `lab2.ml` file probably won't compile.)

## 1. Desugaring `let`

**[10 points]**

Desugar the following (nonrecursive) `let` expressions to the equivalent `fun` expressions applied to arguments. You do not need to evaluate the resulting `fun` expressions. Use the OCaml interpreter to test that your desugared versions are equivalent to the original versions. (You don't have to prove this to anyone but yourself.)

> **Note**
>
> A non-recursive `let` / `and` form binds multiple values to the result of evaluating the corresponding expressions, but **none of the binding expressions can depend on the other bindings**. (In a *recursive* `let rec` / `and` form, any or all of the binding expressions can depend on the bindings.) A non-recursive `let` / `and` form is therefore equivalent to a function of more than one argument applied to its arguments, as discussed in the lectures.

### a.

```
let x = 20
and y = 2 * 4
in x * (2 + y)
```

### b.

```
let a = 1.0
and b = 20.0
and c = 3.0
in sqrt (b *. b -. 4.0 *. a *. c)
```

### c.

For this problem, desugar all of the `let` expressions. Note that successive `let`/`in` forms are *not* the same as a `let`/`and` form with multiple `and`s, because expressions can depend on earlier bindings.

```
let x = 1 in
let y = 2 in
let z = 3 in
  x * y * z
```

### d.

For this problem, desugar all of the `let` expressions.

```
let x = 1 in
let x = 2 in
let x = 3 in
  x * x * x
```

## 2. Desugaring `let` and the substitution model

**[5 points]**

Using the substitution model (including desugaring `let` to `fun`), evaluate the following expression. You may skip obvious steps (for instance, you can reduce `2 + 2` to `4` in a single step).

> **Hint**
>
> Desugar all the `let`s to `fun`s before doing anything else.

Our evaluation took about 35 lines. Watch out for lambda shielding!

```
let x = 2 * 10
and y = 3 + 4
in
  let y = 14 in
  let z = 22 in
    x * y * z
```

## 3. Why doesn't this work?

**[5 points]**

Ben Bitfiddle can't understand why the following code gives an error:

```
let x = 10
and y = x * 2
and z = y + 3
in x + y + z
```

When Ben runs this (expecting the result to be 53), OCaml complains about `x` being an unbound value. "That's not true!" cries Ben angrily, "`x` was bound on the first line!". Explain why this doesn't work by first desugaring the `let` to a `fun`, and then explain in words why it can't work, by referring to the way expressions get evaluated (you don't need to evaluate the expression explicitly). Then show Ben a simple way to fix this code to make it do what he wants.