Part B: Evaluation (35 points)

In this section, write all essay-question-type answers inside OCaml comments.

1. (SICP exercise 1.5)

[10 points]

> **Note**
>
> Before tackling this problem, read the subsection in SICP, section 1.1.5 (scroll down a little to get to the subsection) called *Applicative order versus normal order* (not covered in class!). Applicative order evaluation is just the evaluation rule we described in class; it's usually just called "strict evaluation". Normal order evaluation is an alternative to the evaluation rule we learned in class. In normal order evaluation, nothing is evaluated unless it needs to be to get the final result (it's sometimes called "call-by-need" where applicative order evaluation is called "call-by-value"). Normal order evaluation is used in some functional languages like Haskell (more accurately, Haskell uses *lazy evaluation*, which is a more efficient version of normal order evaluation).

Ben Bitfiddle has invented a test to determine whether the interpreter he is faced with is using applicative order evaluation or normal order evaluation. He defines the following two functions:

```
let rec p () = p ()
let test x y = if x = 0 then 0 else y
```

He then evaluates the expression:

```
test 0 (p ())
```

What behavior will Ben observe with an interpreter that uses applicative order evaluation? What behavior will he observe with an interpreter that uses normal order evaluation? Explain your answer. (Assume that the evaluation rule for the special form `if` is the same whether the interpreter is using normal or applicative order: the predicate (test) expression is evaluated first,

and the result determines whether to evaluate the consequent (`then`) or the alternative (`else`) expression.)

NOTE: This problem doesn't require a lengthy explanation; two or three sentences should be enough.

2. (SICP exercise 1.6)

[5 points]

This problem is one of my (Mike's) favorites.

Alyssa P. Hacker doesn't see why `if` needs to be provided as a special syntactic form. "Why can't I just define `if` as an ordinary function?" she asks. Alyssa's friend Eva Lu Ator claims this can indeed be done, and she defines a new version of `if` using pattern matching[1]:

```
let new_if predicate then_clause else_clause =
  match predicate with
    | true  -> then_clause
    | false -> else_clause
```

> **Note**
>
> Of course, since this is a function it will have to be called using function syntax (the `if` syntax is built-in to OCaml).

Eva demonstrates its use to Alyssa:

```
# new_if (2 = 3) 0 5;;
- : int = 5
# new_if (1 = 1) 0 5;;
- : int = 0
```

Delighted, Alyssa uses `new_if` to write the following program to compute square roots:

```
let square x = x *. x
let average x y = (x +. y) /. 2.0

let improve guess x = average guess (x /. guess)
let is_good_enough guess x =
  abs_float (square guess -. x) < 0.00001
```

```
let rec sqrt_iter guess x =
  new_if (is_good_enough guess x)
         guess
         (sqrt_iter (improve guess x) x)
```

What happens when Alyssa attempts to use this to compute square roots? Explain.

3. (SICP exercise 1.9)

[20 points]

Each of the following two functions defines a method for adding two positive integers in terms of the functions `inc`, which increments its argument by 1, and `dec`, which decrements its argument by 1.

```
let rec add_a a b =
  if a = 0
    then b
    else inc (add_a (dec a) b)

let rec add_b a b =
  if a = 0
    then b
    else add_b (dec a) (inc b)
```

Note that `inc` or `dec` could trivially be defined as:

```
let inc a = a + 1
let dec a = a - 1
```

but for this problem, assume that they are primitive functions.

We will use the substitution model to understand the process generated by each function when evaluating `add_a 2 5` and `add_b 2 5`.

1. Both `add_a` and `add_b` are recursive functions. State whether they generate recursive or iterative *processes* in the sense described in the lectures and in the book.
2. For the `add_a` function only, we want you to write out the substitution model evaluation in great detail. That means you have to actually write out the fact that numbers evaluate to themselves, built-in functions evaluate to their internal representations, functions with arguments desugar to their corresponding `fun` forms, *etc.* Don't skip steps or you'll lose marks on this problem. Also note when names are bound to their values.

One shortcut that you can take is to replace the body of a function with ellipses (`...`), or to replace parts of it that aren't relevant with ellipses. Please indent your work to make it obvious when you are evaluating a subexpression, a sub-sub-expression, *etc.* Be explicit about writing out evaluate, apply, and substitution steps, and when you are desugaring a function definition into the equivalent `fun` form. Also note where you are invoking a special form rule distinct from the usual evaluation rule (*e.g.* with an `if` expression).

Assume that function arguments evaluate from left to right. This isn't necessary to get the right result, but it will make it easier for your graders to grade if everyone does this the same way.

Write the substitution model evaluation in an OCaml comment. Our solution for this part is around 60-70 lines long. If your solution is much shorter than that, then you are skipping too many steps.

You may find the reading on the substitution model to be useful.

> **Note**
>
> We realize that some of you (OK, *all* of you) may dislike this problem. Think of it the same way you might think of taking cod liver oil or eating broccoli; unpleasant but ultimately good for you. Understanding how a computer evaluates expressions is fundamental knowledge, and we will revisit this idea several times in this course (but it will never again be as tedious as in this problem!).

3. For the `add_b` function, we want you to correct the following substitution model evaluation that Ben Bitfiddle dashed off in a hurry. Copy his evaluation into a comment and add the lines that Ben forgot to include. Each such line should start with the characters:

```
>>>
```

so your grader can easily identify them.

> **Note**
>
> Don't write out your own substitution model evaluation from scratch! Just modify the evaluation below by adding the missing lines.

```
(*
let rec add_b a b =
  if a = 0
```

```
     then b
     else add_b (dec a) (inc b)

Desugar this to:

let rec add_b =
  fun a b ->
    if a = 0
       then b
       else add_b (dec a) (inc b)

Bind the name "add_b" to the value:

  fun a b ->
    if a = 0
       then b
       else add_b (dec a) (inc b)

Evaluate (add_b 2 5)
  apply (fun a b -> if ...) to 2, 5
  substitute 2 for a, 5 for b in (if ...)
    -> if 2 = 0 then 5 else add_b (dec 2) (inc 5)
  evaluate (if 2 = 0 then 5 else add_b (dec 2) (inc 5))
    if is a special form, so evaluate the first operand:
      evaluate (2 = 0)
        apply = to 2, 0 -> false
    first argument of if is false, so evaluate the third operand:
      evaluate (add_b (dec 2) (inc 5))
        evaluate (dec 2)
          apply dec to 2 -> 1
        evaluate (inc 5)
          apply inc to 5 -> 6
        apply (fun a b -> if ...) to 1, 6
        substitute 1 for a, 6 for b in (if ...)
          -> if 1 = 0 then 6 else add_b (dec 1) (inc 6)
        evaluate (if 1 = 0 then 6 else add_b (dec 1) (inc 6))
          if is a special form, so evaluate the first operand:
            evaluate (1 = 0)
              apply = to 1, 0 -> false
          first argument of if is false, so evaluate the third operand:
            evaluate (add_b (dec 1) (inc 6))
              evaluate (dec 1)
                apply dec to 1 -> 0
              evaluate (inc 6)
                apply inc to 6 -> 7
              apply (fun a b -> if ...) to 0, 7
              substitute 0 for a, 7 for b in (if ...)
                -> if 0 = 0 then 7 else add_b (dec 0) (inc 7)
              evaluate (if 0 = 0 then 7 else add_b (dec 0) (inc 7))
                if is a special form, so evaluate the first operand:
                  evaluate (0 = 0)
                    apply = to 0, 0 -> true
                first argument of if is true, so evaluate the second operand:
```

```
                    result: 7
*)
```

---

1. OCaml's pattern matching syntax was covered in recitation lecture 1. ↵