Part A: Exercises (35 points)

Some of the following problems are taken from the textbook (Structure and Interpretation of Computer Programs, or SICP for short). If so, the SICP exercise numbers are included, though you shouldn't need to consult the book.

The online version of SICP is here.

0. In-class exercise [**No collaboration**]

> **Note**
>
> Problems in the in-class exercises section (there's only one in this assignment) were done in class collaboratively. However, you *must* reproduce them here (or write them from scratch, if you didn't attend class) *without* any collaboration. You are allowed to use notes/code that **you yourself** wrote down during the lecture, but that is all.
>
> You also cannot ask the TAs or the instructor for help solving in-class exercises, unless you are working on a rework of the (graded) assignment which included the exercise(s).
>
> See the collaboration policies for more information on in-class exercises.

[5 points]

Write a recursive function called `sum_squares_to` that takes a non-negative integer `n` as its only argument. It returns the sum of the squares of all integers from 0 to `n`.

You can use either an `if` expression or pattern matching.

Examples

```
sum_squares_to 0    --> 0
sum_squares_to 1    --> 1
sum_squares_to 2    --> 5
sum_squares_to 3    --> 14
sum_squares_to 5    --> 55
sum_squares_to 10   --> 385
sum_squares_to 20   --> 2870
sum_squares_to 50   --> 42925
sum_squares_to 100  --> 338350
sum_squares_to 1000 --> 333833500
```

1. Expressions

[10 points]

Below is a sequence of expressions. What is the result (the type and value or the error message) printed by the OCaml interpreter in response to each expression? Assume that the sequence is to be evaluated in the order in which it is presented. If the interpreter indicates an error, explain briefly (one sentence) why the error occurred. There are also some other questions below which you should answer to the best of your ability. Note that entering each code fragment interactively requires that you add the `;;` terminator to terminate input. Write your answers as OCaml comments `(* like this *)`.

1. `10`
2. `10.`
3. `5 + 3 + 4`
4. `3.2 + 4.2`
5. `3 +. 4`
6. `3 + 4.2`
7. `3 +. 4.2`
8. `3.0 +. 4.2`
9. `9 - 3 - 1`
10. `9 - (3 - 1)`
11. `let a = 3`
12. `let b = a + 1`

13. `a = b`
14. `[1; 2; 3] = [1; 2; 3]`
15. `[1; 2; 3] == [1; 2; 3]` Is this the same as or different from the previous expression? Why or why not?
16. `[(1, 2, 3)]`
17. `[1, 2, 3]` Explain why this gives the result it does. This is a nasty pitfall which highlights one of the less desirable features of OCaml's syntax. (See the OCaml cheat sheet for more on this.)
18. `if b > a && b < a * b then b else a`
19. `if b > a and b < a * b then b else a`
20. `2 + if b > a then b else a`
21. `if b > a then b else a + 2` Why is this different from the previous case?
22. `(if b > a then b else a) + 2`
23. `if b > a then b` This is not a syntax error. Why does this give a type error? *Hint*: What does OCaml assume if the `else` in an `if` / `then` / `else` form is left off?

2. (SICP exercise 1.3)

[10 points]

Define a function that takes three integer numbers as arguments and returns the sum of the squares of the two larger numbers. Call the function you define `sum_of_squares_of_two_largest`. You will probably find the `&&` special operator to be handy.

3. (SICP exercise 1.4)

[10 points]

Our evaluation model allows you to use a function which is a compound expression (something that evaluates to a function). Use this observation to describe the behavior of the following function:

```
let a_plus_abs_b a b =
  (if b > 0 then (+) else (-)) a b
```

Write your answer in a comment. Note that surrounding an operator with parentheses makes it into a two-argument function, so

```
(+) 2 3
```

is the same as:

```
2 + 3
```