Part C: Recursion (30 points)

1. Computing *e*

[10 points]

In this problem we're going to write a function that enables us to compute the number *e*, the base of natural logarithms, which is equal to 2.7182818... We will do this by summing a part of an infinite series expansion which computes *e*:

```
e = 1/0! + 1/1! + 1/2! + ...
```

where `n!` is the factorial of `n`, which in defined as

```
n! = n * (n-1) * (n-2) * ... * 1
```

We'll start out with the factorial function itself:

```
(* This function computes the factorial of the input number,
   which for a number n is equal to n * (n-1) * ... * 1. *)
let rec factorial n =
  if n = 0 then 1 else n * factorial (n - 1)
```

Note

The definition of `factorial` has to be included in your assignment submission or the rest of the code won't work.

Also, if you're unhappy because this is a space-inefficient way to compute factorials, you're right! But it won't matter for this problem.

a. `e_term`

Write a simple function called `e_term` which takes a (non-negative) integer argument and computes that term of the infinite series expansion of `e`. This function is not recursive. Note that the result must be a floating-point number; use the `float_of_int` function to convert from an OCaml `int` to a `float`.

Note

OCaml never implicitly promotes one numeric type to another, so you must do it explicitly if that's what you want.

Examples

```
# e_term 0;;
- : float = 1.
# e_term 1;;
- : float = 1.
# e_term 2;;
- : float = 0.5
# e_term 5;;
- : float = 0.00833333333333333322
```

b. `e_approximation`

Write a recursive function called `e_approximation` that takes one positive integer argument and computes an approximation to *e* (an "e-proximation", as it were) by summing up that many terms of the infinite series expansion of *e* (actually, it'll sum up the first n+1 terms, since it starts at term 0 and ends at term n). **Write the function as a linear recursive process.** Use your `e_term` function to help you write this one.

Examples

```
# e_approximation 0;;
- : float = 1.
# e_approximation 1;;
- : float = 2.
# e_approximation 2;;
- : float = 2.5
# e_approximation 5;;
- : float = 2.71666666666666634
# e_approximation 10;;
- : float = 2.71828180114638451
```

c. Computing the approximation

Compute an approximation to *e* by summing up to the 20th term of the infinite series expansion. Write down the answer that OCaml gives you in a comment in your lab submission. Then write down the value of `exp 1.0` which is `e` to the power of `1`, and compare with the result given by `e_approximation 20` (they should be nearly identical).

d. Too many terms?

What happens if you try to compute a better approximation to `e` by summing up to the 100th term of the infinite series expansion? Why does this happen? Write your answer in a comment.

2. Mutual recursion

[5 points]

It's possible to have a kind of recursion which involves more than one function; this is called *mutual recursion*. A simple example is a pair of functions `is_even` and `is_odd`. `is_even` is a predicate which returns `true` if its argument is an even (non-negative) integer and `false` if its argument is an odd (non-negative) integer (zero is considered even). `is_odd` returns `true` if its numeric argument is odd and `false` otherwise. Write these functions, using only recursion, testing for equality with zero, and subtracting 1. (The solution is very short.) Since the solution uses mutual recursion, the `is_even` function will need to call the `is_odd` function and the `is_odd` function will need to call the `is_even` function.

Note

Of course, this is an extremely inefficient way of computing evenness and oddness! It's just for illustration purposes.

When defining mutually recursive functions in OCaml, you need to use the `let rec ... and ...` syntax:

```
let rec f1 a b c = ...   (* ... may include f2 *)
and f2 x y z = ...       (* ... may include f1 *)
```

This works for any number of mutually-recursive functions. The whole form starts with a `let rec` and then each function after the first in the group starts with `and`.

Note

Mutual recursion might seem like something extremely esoteric with few real uses, but it isn't. If you are writing a programming language interpreter or compiler (as you will if you take CS 131 or CS 164), you will find that your code is full of large swaths of mutually-recursive functions.

3. (SICP exercise 1.11)

[5 points]

A function `f` is defined by the rules:

- `f(n) = n` if `n < 3`
- `f(n) = f(n - 1) + 2 * f(n - 2) + 3 * f(n - 3)` if `n >= 3`.

Write a function called `f_rec` that computes `f` by means of a recursive process. Write another function called `f_iter` that computes `f` by means of an iterative process.

Hint

The recursive definition is straightforward. Use it to check the correctness of the iterative definition. The iterative definition will need a helper function that keeps track of the last three numbers in the series, among other things.

Examples

```
# f_rec 0;;
- : int = 0
# f_rec 1;;
- : int = 1
# f_rec 3;;
- : int = 4
# f_rec 10;;
- : int = 1892
# f_rec 20;;
- : int = 10771211
(* f_iter gives the same results *)
```

4. (SICP exercise 1.12)

[10 points]

The following pattern of numbers is called "Pascal's triangle":

```
    1
   1 1
  1 2 1
 1 3 3 1
1 4 6 4 1
```

The numbers at the edges of the triangle are all 1, and each number inside the triangle is the sum of the two numbers above it to the left and right. Write a function called `pascal_coefficient` which takes two `int` arguments (corresponding to the row number (starting from 1) and the index inside the row (also starting from 1)) and computes elements of Pascal's triangle by means of a recursive process.

Use pattern matching to make the code cleaner. **This is not optional!** Use of `if` / `then` / `else` in your solution will result in lost marks.

Note

If you have a `match`, but use a `when` clause with each match case, this is equivalent to having nested `if` / `then` / `else` expressions, and you will still lose marks. See the hint below.

For arguments that don't correspond to locations in Pascal's triangle (like numbers < 1 or index numbers that are greater than the row number) you should signal an error using the code: `failwith "invalid arguments"`. This raises a `Failure` exception. We'll talk more about exceptions later in the course.

Examples

```
# pascal_coefficient 1 1 ;; (* row 1, index 1 in row *)
- : int = 1
# pascal_coefficient 2 1 ;;
- : int = 1
# pascal_coefficient 2 2 ;;
- : int = 1
# pascal_coefficient 3 1 ;;
- : int = 1
# pascal_coefficient 3 2 ;;
- : int = 2
# pascal_coefficient 3 3 ;;
- : int = 1
# pascal_coefficient 10 5 ;;
- : int = 126
# pascal_coefficient 1 0 ;;
Exception: Failure "invalid arguments".
```

Hints

- It's much easier to write this as a recursive process (*i.e.* with pending operations) than as an iterative process, though both approaches will involve recursive functions. Since you don't have to do it both ways, we recommend you write it as a recursive process.
- Pattern match on a tuple of both input arguments. Use the `when` form in a pattern when you need to specify *non-structural* conditions for a match (like two things being equal). **Only use `when` when you can't use structural pattern matching.** For instance, don't do this kind of thing:

  ```
  match x with
    | x' when x' = 1 -> ...
    ...
  ```

  when you can do this instead:

  ```
  match x with
    | 1 -> ...
    ...
  ```

  Also don't forget about the `_` (wildcard) syntax for don't-care patterns. Remember that a number in a pattern matches that literal number only.
- It's convenient (though not required) to use the first pattern match for error checking only.

Despite all this explanation, this function only needs to be a few lines long.