Part A: Working with lists (50 points)

0. In-class exercise

[5 points]

Write a function called `range` which takes two `int` arguments `m` and `n`, and returns a list containing all the numbers from `m` up to an including `n`. This function should generate a recursive process. Also write a function called `irange` which generates an iterative process.

Examples

Both functions (`range` and `irange`) return the same results for the same inputs.

```
# range 1 10;;
- : int list = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
# range 1 1;;
- : int list = [1]
# range 1 0;;
- : int list = []
```

You may find the library function `List.rev` (which returns the reverse of a list) to be useful in your `irange` definition, though it's not absolutely necessary.

Note

Both `range` and `irange` must have linear time complexities, *i.e.* they must be $\Theta(|n - m|)$
.

1. (SICP exercise 2.17)

[5 points]

Define a function `last_sublist` that returns the list that contains only the last element of a given (nonempty) list:

```
# last_sublist [23; 72; 149; 34] ;;
- : int list = [34]
```

Your function should signal an error if the list is empty. Use `invalid_arg` to generate the error; the error message should be `"last_sublist: empty list"` (that *exact* message; we'll check it!).

You should write this function using pattern matching and the `function` keyword, since there is only one argument and the function pattern matches on it. The resulting function should generate a linear iterative process when run (in other words, the function should be tail recursive.) Also, this function should work on arbitrary lists (the element type shouldn't matter).

2. (SICP exercise 2.18)

[5 points]

Define a function `reverse` that takes a list as argument and returns a list of the same elements in reverse order:

```
# reverse [1; 4; 9; 16; 25] ;;
- : int list = [25; 16; 9; 4; 1]
# reverse [] ;;
- : 'a list = []
# reverse [[1; 4]; [9]; [16; 25]] ;;
- : int list list = [[16; 25]; [9]; [1; 4]]
```

Note the type of the last example; it is a `list` of `int list`s! Read `int list list` as `(int list) list`.

Your `reverse` function should have a linear time complexity, be tail recursive (*i.e.* iterative), and should (naturally) work on lists of any element types. It should *not* use the list append (`@`) operator.

There is a library function called `List.rev` which reverses lists. Obviously, you shouldn't use it in your answer to this problem.

3. (SICP exercise 2.21)

[5 points]

The function `square_list` takes a list of integers as argument and returns a list of the squares of those numbers.

```
# square_list [1; 2; 3; 4]
- : int list = [1; 4; 9; 16]
```

Here are two different definitions of `square_list`. Complete both of them by filling in the missing expressions:

```
let rec square_list = function
  | [] -> []
  | h :: t -> <??>

let square_list2 items = List.map <??> <??>
```

Use the List library documentation to find the description of `List.map`.

4. (SICP exercise 2.22)

[5 points]

Louis Reasoner tries to rewrite the first `square_list` function from the previous problem so that it evolves an iterative process:

```
let square_list items =
  let rec iter things answer =
    match things with
      | [] -> answer
      | h :: t -> iter t ((h * h) :: answer)
  in iter items []
```

Unfortunately, defining `square_list` this way produces the answer list in the reverse order of the one desired. Why? Write your answer in a comment.

Louis then tries to fix his bug by interchanging the arguments to the `::` constructor:

```
let square_list items =
  let rec iter things answer =
    match things with
      | [] -> answer
      | h :: t -> iter t (answer :: (h * h))
  in iter items []
```

This doesn't work either. Explain why in a comment.

Can you modify Louis' second solution slightly to make it work properly? (By "slightly", we mean changing the `::` constructor to a different list operator and making one more small modification on the same line?) If so, would the resulting function be efficient? Why or why not?

Note

We’re interested in time efficiency here, not space efficiency.

5. `count_negative_numbers`

[**5 points**]

Write a function called `count_negative_numbers` that counts the negative integers in a list and returns the count.

6. `power_of_two_list`

[**5 points**]

Write a function called `power_of_two_list` that takes in an integer `n`, and creates a list containing the first `n` powers of 2 starting with 2^0 and up to 2^{n-1}.

You can write a `pow` helper function inside this function if you like (though you’re not required to; there are other ways to do it). The `pow` function (if you write it) takes two non-negative integers and returns the first raised to the power of the second. (For the pedantic: you can assume that $0^0 = 1$). Don’t use the `**` operator (which is a floating-point operator anyway) in your definition of `pow`.

Examples

```
# power_of_two_list 0 ;;
- : int list = []
# power_of_two_list 1 ;;
- : int list = [1]
# power_of_two_list 2 ;;
- : int list = [1; 2]
# power_of_two_list 4 ;;
- : int list = [1; 2; 4; 8]
# power_of_two_list 8 ;;
- : int list = [1; 2; 4; 8; 16; 32; 64; 128]
```

7. `prefix_sum`

[5 points]

Write a function called `prefix_sum` that takes in a list of numbers, and returns a list containing the prefix sum of the original list. *e.g.* `prefix-sum [1; 3; 5; 2] ==> [1; 4; 9; 11]`. The prefix sum is the sum of all of the elements in the list up to that point, so for the list `[1; 3; 5; 2]` the prefix sum is `[1; 1+3; 1+3+5; 1+3+5+2]` or `[1; 4; 9; 11]`.

Examples

```
# prefix_sum [] ;;
- : int list = []
# prefix_sum [1] ;;
- : int list = [1]
# prefix_sum [1; 2; 3; 4; 5] ;;
- : int list = [1; 3; 6; 10; 15]
# prefix_sum [-1; 1; -1; 1; -1; 1] ;;
- : int list = [-1; 0; -1; 0; -1; 0]
```

8. (SICP exercise 2.27)

[5 points]

Modify the `reverse` function you defined previously in this assignment to produce a `deep_reverse` function that takes a list of lists (of arbitrary type) as its argument and returns as its value the same list with its elements reversed and with its immediate sublists reversed as well.

Note

You are allowed to use the `reverse` function you defined above in your `deep_reverse` function.

Examples

```
# let lst = [[1; 2]; [3; 4]] ;;
val lst : int list list = [[1; 2]; [3; 4]]
# reverse lst ;;
- : int list list = [[3; 4]; [1; 2]]
# deep_reverse lst ;;
- : int list list = [[4; 3]; [2; 1]]
# let lst2 = [[[1; 2]; [3; 4]]; [[5; 6]; [7; 8]]] ;;
- : int list list list = [[[1; 2]; [3; 4]]; [[5; 6]; [7; 8]]]
# deep_reverse lst2 ;;
- : int list list list = [[[7; 8]; [5; 6]]; [[3; 4]; [1; 2]]]
```

9. Nested lists

[5 points]

Sometimes people learning OCaml from a background of dynamically typed languages like Python miss having lists which can store arbitrary values. In particular, it would be nice to have a list that can store either values of a particular type `'a`, or a list of such values, or a list of lists of such values, and so on, with any combination of values and lists. This is easily modeled in OCaml by defining a new datatype:

```
type 'a nested_list =
  | Value of 'a
  | List of 'a nested_list list
```

Now you can have a list-like data structure that can mix together values and lists with arbitrary nesting. (This datatype is actually isomorphic to a datatype called *S-expressions* which are used as the basis of the syntax of Scheme and related languages.) For instance:

```
# Value 10 ;;
- : int nested_list = Value 10
# List [Value 10] ;;
- : int nested_list = List [Value 10]
# List [Value 10; Value 20; Value 30] ;;
- : int nested_list = List [Value 10; Value 20; Value 30]
# List [Value 10; List [Value 20; List [Value 30; Value 40]; Value 50]; Value 60]
;;
- : int nested_list =
List
 [Value 10; List [Value 20; List [Value 30; Value 40]; Value 50]; Value 60]
# List [List [Value 1; Value 2]; List [Value 3; Value 4]] ;;
- : int nested_list = List [List [Value 1; Value 2]; List [Value 3; Value 4]]
```

Define a version of the `deep_reverse` function from the previous problem that works on `nested_list`s. Call it `deep_reverse_nested`. Don't use the `reverse` or `deep_reverse` functions in your definition. Note that `Value`s don't get reversed, because there is in general no way to reverse them; only the `List` constructor contents get reversed. `deep_reverse_nested` should reverse lists all the way down, no matter how deeply nested the lists are.

Examples

```
# deep_reverse_nested (Value 10) ;;
- : int nested_list = Value 10
# deep_reverse_nested (List [Value 10; Value 20; Value 30; Value 40]) ;;
- : int nested_list = List [Value 40; Value 30; Value 20; Value 10]
# deep_reverse_nested (List [List [Value 10; Value 20]; List [Value 30;
Value 40]]) ;;
- : int nested_list =
List [List [Value 40; Value 30]; List [Value 20; Value 10]]
# deep_reverse_nested (List [Value 10; List [Value 20; Value 30]]) ;;
- : int nested_list = List [List [Value 30; Value 20]; Value 10]
# deep_reverse_nested (List [List [Value 10; Value 20]; Value 30]) ;;
- : int nested_list = List [Value 30; List [Value 20; Value 10]]
# deep_reverse_nested (List [Value 10; List [Value 20; List [Value 30;
Value 40]; Value 50]; Value 60]) ;;
- : int nested_list =
List
 [Value 60; List [Value 50; List [Value 40; Value 30]; Value 20]; Value 10]
```

Hint

Check for the `Value` case first, because if the input is just a `Value`, it doesn’t need to be reversed. Otherwise, extract the (OCaml) list from the `List` constructor and pass it to a recursive helper function which will assemble the result.

Warning

There is a very simple solution to this problem that uses the library functions `List.map` and `List.rev`. This is *not* what we want, so please don't use those functions in your answer.