Part C: Some harder problems with lists (30 points)

1. (SICP exercise 2.32)

[5 points]

We can represent a *set* as a list of distinct elements, and we can represent the set of all subsets of the set as a list of lists. For example, if the set is `[1; 2; 3]`, then the set of all subsets is `[[]; [3]; [2]; [2; 3]; [1]; [1; 3]; [1; 2]; [1; 2; 3]]`. Complete the following definition of a function that generates the set of subsets of a set and give a clear explanation of why it works:

```
let rec subsets = function
  | [] -> [[]]
  | h :: t -> let rest = subsets t in
      rest @ (List.map <??> rest)
```

This problem is a classic. Note that the order of elements in a set is unimportant, so the list results can come in any order (but don't duplicate anything!). (Using lists for sets is not optimal design, but we will revisit this question later.)

Don't forget the "clear explanation of how it works"! Put this in an OCaml comment. (It doesn't have to be very long.)

Examples

```
# subsets [] ;;
- : 'a list list = [[]]
# subsets [1] ;;
- : int list list = [[]; [1]]
# subsets [1;2;3] ;;
- : int list list = [[]; [3]; [2]; [2; 3]; [1]; [1; 3]; [1; 2]; [1; 2; 3]]
```

2. (SICP exercise 2.33)

[10 points]

This is another classic problem (SICP is full of them!).

Fill in the missing expressions to complete the following definitions of some basic list-manipulation operations as accumulations:

```
let rec accumulate op initial sequence =
  match sequence with
    | [] -> initial
    | h :: t -> op h (accumulate op initial t)

let map p sequence =
  accumulate (fun x r -> <??>) [] sequence

let append seq1 seq2 =
  accumulate (fun x r -> x :: r) <??> <??>

let length sequence =
  accumulate <??> 0 sequence
```

Note

The `accumulate` function is so generally useful that it is a library function in nearly all functional languages; in OCaml it's called `List.fold_right`, though the arguments are in a different order.

Tip

The `op` part of the problem is the most important part. Each `op` function must be a function of two arguments, the first being the current list value being looked at, and the second the rest of the list after being recursively processed by the `accumulate` function. Also, don't assume that the initial value is always *e.g.* the empty list. The missing parts are very short.

3. (SICP exercise 2.36)

[5 points]

The function `accumulate_n` is similar to `accumulate` except that it takes as its third argument a list of lists, which are all assumed to have the same number of elements. It applies the designated accumulation function to combine all the first elements of the lists, all the second elements of the lists, and so on, and returns a list of the results. For instance, if `s` is a list containing four lists, `[[1;2;3];[4;5;6];[7;8;9];[10;11;12]]`, then the value of `accumulate_n (+) 0 s` should be the list `[22;26;30]`. Fill in the missing expressions in the following definition of `accumulate_n`:

```
let rec accumulate_n op init seqs =
  match seqs with
    | [] -> failwith "empty list"
    | [] :: _ -> []   (* assume all sublists are empty *)
    | _ ->  (* non-empty list containing non-empty sublists *)
        accumulate op init <??> :: accumulate_n op init <??>
```

Examples

```
# accumulate_n ( + ) 0 [[];[];[]] ;;
- : int list = []
# accumulate_n ( + ) 0 [[1;2;3];[4;5;6];[7;8;9];[10;11;12]] ;;
- : int list = [22; 26; 30]
# accumulate_n ( * ) 1 [[2;3];[4;5]] ;;
- : int list = [8; 15]
```

Hint

`map` (or `List.map` if you prefer) is your friend. You may also find the `List.hd` (head) and `List.tl` (tail) functions to be useful. Again, the parts you need to fill in are very short .

4. (SICP exercise 2.37)

[10 points]

Suppose we represent vectors `v = (vi)` as lists of numbers, and matrices `m = (mij)` as lists of vectors (the rows of the matrix). For example, the 3x4 matrix:

```
+---------+
| 1 2 3 4 |
| 4 5 6 6 |
| 6 7 8 9 |
+---------+
```

is represented as the list of lists `[[1;2;3;4];[4;5;6;6];[6;7;8;9]]`. With this representation, we can use list operations to concisely express the basic matrix and vector operations. These operations (which are described in any book on matrix algebra) are the following:

```
dot_product v w
  (* returns: the number d, where d = sum_i (v_i * w_i) *)

matrix_times_vector m v
  (* returns: the vector t, where t_i = sum_j (m_ij * v_j) *)

matrix_times_matrix m n
  (* returns: the matrix p, where p_ij = sum_k (m_ik * n_kj) *)

transpose m
  (* returns: the matrix n, where n_ij = m_ji *)
```

We can define the dot product as:

```
let dot_product v w = accumulate (+) 0 (map2 ( * ) v w)
```

where `map2` is a version of `map` which maps a two-argument function (or operator) over *two* lists of equal lengths, returning a list of that length.

Fill in the missing expressions in the following functions for computing `map2` and the other matrix operations. (The function `accumulate_n` was defined in the previous problem.)

```
let rec map2 f x y =
  match (x, y) with
    | ([], []) -> []
    | ([], _) -> failwith "unequal lists"
    | (_, []) -> failwith "unequal lists"
    | <??>

let matrix_times_vector m v = map <??> m

let transpose mat = accumulate_n <??> <??> mat

let matrix_times_matrix m n =
  let cols = transpose n in
     map <??> m
```

Examples

```
# dot_product [] [] ;;
- : int = 0
# dot_product [1;2;3] [4;5;6] ;;
- : int = 32
# matrix_times_vector [[1;0];[0;1]] [10;20] ;;
- : int list = [10; 20]
# matrix_times_vector [[1;2];[3;4]] [-2;3] ;;
- : int list = [4; 6]
# transpose [[1;2];[3;4]] ;;
- : int list list = [[1; 3]; [2; 4]]
# transpose [[1;2;3];[4;5;6]] ;;
- : int list list = [[1; 4]; [2; 5]; [3; 6]]
# matrix_times_matrix [[1;0];[0;1]] [[1;2];[3;4]] ;;
- : int list list = [[1; 2]; [3; 4]]
# matrix_times_matrix [[1;2];[3;4]] [[1;2];[3;4]] ;;
- : int list list = [[7; 10]; [15; 22]]
# matrix_times_matrix [[1;2;3];[4;5;6]] [[1;2];[3;4];[5;6]] ;;
- : int list list = [[22; 28]; [49; 64]]
```

Tip

You can use the solutions of some of the functions in later functions. The missing parts are quite short, so don’t do anything complicated! Finally, realize that multiplying a matrix by a matrix can be decomposed into multiplying each row of the first matrix by the entire second matrix.