CISS445: Programming Languages
Assignment 5

Objectives: This is the first of several OCAML assignments where the main object is to help you learn the basic OCAML language, including basic types and operators, tuples and lists with their operations, declarations, recursion, and matchings.

In this assignment, several questions must be completed using tail recursion. In these cases, you will need to write a helper function that performs a recursion. This helper function will need to carry one or more extra variables (sometimes called accumulators). In this case, you should hide the helper function within the actual function you want to write using the let-in syntax. You must also ensure that your implementation is reasonably efficient. Note that for a tail recursion, the result of the original function call is computed during the last recursive call (and is usually the value inside an accumulator) and not during the last return from all recursive calls. (Refer to your notes).

You may include any code developed in previous assignments. From now on, I reserve the right to modify or include other test cases.

No discussion is allowed. Talk to me if you need help.

Q1. Write a function `seteq` that returns `true` if two lists contains the same elements.

```
Tests                  Correct values
seteq [] []            true
seteq [1] []           false
seteq [] [1]           false
seteq [1] [1]          true
seteq [1] [1;1]        true
seteq [1;1] [1]        true
seteq [1;2] [2;1]      true
seteq [2;1] [1;2]      true
seteq [1;2] [3;2;1]    false
seteq [1;2] [3;3;2;1]  false
```

Q2. Write a function `setsimplify` that removes duplicate elements from a list as a set. The relative position of the elements in the set must be retained. You must implement this using tail recursion.

```
Tests                          Correct values
setsimplify []                 []
setsimplify [2;1]              [2;1]
setsimplify [2;2;1]            [2;1]
setsimplify [2;1;1;1]          [2;1]
setsimplify [3;1;1;2;1;1;1;3] [3;1;2]
```

Q3. Write a function setintersect that returns the intersection of two lists as sets, i.e., the list of elements in both lists. Duplicates must be removed. You must implement this using tail recursion.

Tests	Correct values
`setintersect [1;2] []`	`[]`
`setintersect [1;2] [3;4]`	`[]`
`setintersect [5;2;1] [2;6;9]`	`[2]`
`setintersect [5;2;5] [2;5;6]`	`[5;2]`
`setintersect [5;2;7] [2;3;5;5;6]`	`[5;2]`
`setintersect [1;3;5;7] [7;5;9]`	`[5;7]`

Q4. Write a function `setunion` that returns the union of two lists as sets. There must not be any duplicates.

Tests	Correct values
`setunion [] []`	`[]`
`setunion [] [1]`	`[1]`
`setunion [1;2;3] [3;2;1]`	`[1;2;3]`
`setunion [1;2;2;3;1] [1;5;5;6]`	`[1;2;3;5;6]`
`setunion [3;3;2;1] [2;3]`	`[3;3;2;1]`

Q5. Write a function `setdiff` that returns the difference of two lists as sets. Duplicates must be removed.

```
Tests                        Correct values
setdiff [1;2;3] []           [1;2;3]
setdiff [] [1;2;3]           []
setdiff [1;2;3] [1]          [2;3]
setdiff [1;2;3] [2]          [1;3]
setdiff [1;2;3] [3]          [1;2]
setdiff [1;2;3] [4]          [1;2;3]
setdiff [1;2;2;3] [3]        [1;2]
setdiff [1;2;3] [3;2;1]      []
setdiff [1;2;3] [4;3;2]      [1]
setdiff [1;2;3] [4;3;2;1;0]  []
```

Q6. Write a function `powerset` that returns the powerset of a list (as a set). The lists in the return list may appear in any order and for each such list, the elements may be in any order, but there must not be any duplicates. You may assume that the list passed in does not have duplicates. You must implement this using tail recursion.

```
Tests                Correct values
powerset []          [[]]
powerset [1]         [[]; [1]] (or [[1], []])
powerset [1;2]       [[]; [1]; [2]; [1;2]] (or [[2], [], [1], [2;1]], etc.)
powerset [1;2;3]     [[]; [1]; [2]; [3]; [1;2]; [1;3]; [2;3]; [1;2;3]]
powerset [1;2;3;4]   [[];
                      [1]; [2]; [3]; [4];
                      [1;2]; [1;3]; [1;4]; [2;3]; [2;4]; [3;4];
                      [1;2;3]; [1;2;4]; [1;3;4]; [2;3;4];
                      [1;2;3;4]]
```

Q7. Write a function subsequence that returns true when a list (as a sequence) is a subsequence of another list (as a sequence). A list ys is a subsequence of another xs if the relative positions of the elements in ys is the same as the relative positions in xs. For example, the list

```
ys = [3;5;2]
```

is a subsequence of

```
xs = [1;3;5;4;2]
```

because in `ys`, `3` appears before `5` which appears before `2`. And this is also the case in `xs`. The following are not subsequences of `xs`:

```
[3;2;5], [3;1], [2;1]
```

Tests	Correct values
subsequence [] [1]	true
subsequence [1] [2]	false
subsequence [1] [2;1]	true
subsequence [1;1] [2;1]	false
subsequence [1;1] [1;2;1]	true
subsequence [1;1] [2;1;1]	true
subsequence [1;1] [1;1;2]	true
subsequence [1;1] [1;2;1;3]	true
subsequence [2;1;3] [0;2;1;0;2;1;3]	true

Q8. Write a function subsequences that returns a list of all possible subsequences of a list (as a sequence) passed into the function. This is the same function as `powerset` except that the list passed in may have duplicates, and any element of the return list is itself a list of elements from the argument passed in with their relative positions retained.

```
Tests                   Correct values
subsequences []         []
subsequences [1]        [[]; [1]] (or [[1], []])
subsequences [1;2]      [[]; [1]; [2]; [1;2]] (or [[2], [], [1], [1;2]], etc.)
subsequences [1;2;3]    [[]; [1]; [2]; [3]; [1;2]; [1;3]; [2;3]; [1;2;3]]
subsequences [1;2;3;4]  [[]; [1]; [2]; [3]; [4]; [1;2]; [1;3]; [1;4]; [2;3];
                         [2;4]; [3;4]; [1;2;3]; [1;2;4]; [1;3;4]; [2;3;4];
                         [1;2;3;4]]
```