Assignment #2: ML Introduction #2
Due: October 2, 11:59pm

Note: Your solution must be submitted in a file named hw2.sml.

Overview
This assignment is intended to continue your introduction to the ML programming language. Specifically, this assignment will explore algebraic data types and higher-order functions.

You must include each of the datatype definitions in your submission.

Again, your solutions may not use mutation.

Part 1
Consider the following datatype that defines a list somewhat analogous to ML's built-in list type.

```ml
datatype 'a List =
    ListNode of ('a * 'a List)
  | EmptyList;
```

- `consList: ('a * 'a List) -> 'a List` - Write the `consList` function on the `'a List` datatype. These are analogous to the `::`, `hd`, and `tl` functions on the built-in lists. Note that `headList` and `tailList` are partial functions (i.e., they will raise an error if applied to an `EmptyList`).

Part 2

- `lengthList: 'a List -> int` - Write the `lengthList` function to compute the length of a `'a List` value.

Part 3

- `mapList: ('a -> 'b) -> 'a List -> 'b List` - Using pattern-matching (instead of the functions from Part 1), write the `mapList` function for the `'a List` datatype. This function is, of course, analogous to the standard `map` for `'a list`.

Part 4
Consider the following datatype defining a binary tree (not necessarily sorted).

```ml
datatype 'a BinTree =
    BinTreeNode of {value: 'a, lft: 'a BinTree, rht: 'a BinTree}
  | EmptyBinTree;
```

- `mapBinTree: ('a -> 'b) -> 'a BinTree -> 'b BinTree` - Write the `mapBinTree` function that, given a binary tree, returns a tree constructed by applying the input function to the value at each node.

Part 5
Consider the following datatype:

```ml
datatype 'a ThingCollection =
    OneThing of ('a * 'a ThingCollection)
  | TwoThings of ('a * 'a * 'a ThingCollection)
  | ManyThings of ('a list * 'a ThingCollection)
  | Nothing;
```
This datatype represents a collection of things; things may be of any type, but a collection can contain only things of the same type.

**countThingsInCollection: 'a ThingCollection → int**

Write the function `countThingsInCollection` that takes a `'a ThingCollection` value and returns an integer count of the number of "things" in that collection.

- `countThingsInCollection Nothing;` 0
- `countThingsInCollection (OneThing (7, Nothing));` 1
- `countThingsInCollection (OneThing (7, ManyThings ([1, 2], TwoThings (1, 2, Nothing))));` 5

**Part 6**

**countOneThingNodes: 'a ThingCollection → int**

Write the function `countOneThingNodes` that takes a `'a ThingCollection` and returns the number of OneThing "nodes" in the collection (i.e., the number of times the OneThing constructor appears in the collection).

- `countOneThingNodes Nothing;` 0
- `countOneThingNodes (OneThing (7, Nothing));` 1
- `countOneThingNodes (OneThing (7, ManyThings ([1, 2], TwoThings (1, 2, Nothing))));` 1
- `countOneThingNodes (OneThing (7, ManyThings ([1, 2], TwoThings (1, 2, OneThing (99, Nothing))));` 2

**Part 7**

**countNodesByPredicate: ('a ThingCollection → bool) → 'a ThingCollection → int**

Write the function `countNodesByPredicate` that generalizes the previous function (`countOneThingNodes`) such that it counts the number of "Thing" nodes, where the type of "Thing" is determined by a predicate passed to the function (i.e., the function determines if a node should be counted). Or, another way to think about it, `countNodesByPredicate` returns the number of `'a ThingCollection` constructors in a given collection that satisfy the given predicate.

For example, using the generalized function, the function for the previous problem could be written as:

```haskell
fun countOneThingNodes things = countNodesByPredicate (fn (OneThing _) => true | _ => false) things;
```

- `countNodesByPredicate (fn Nothing => true | _ => false) Nothing;` Nothing
- `countNodesByPredicate (fn (OneThing _) => true | _ => false) Nothing;` 0
- `countNodesByPredicate (fn (OneThing _) => true | _ => false) (OneThing (7, Nothing));` 1
- `countNodesByPredicate (fn (OneThing (7, _)) => true | _ => false) (OneThing (7, ManyThings ([1, 2], TwoThings (1, 2, Nothing))));` 1
- `countNodesByPredicate (fn (OneThing (2, _)) => true | _ => false) (OneThing (7, ManyThings ([1, 2], TwoThings (1, 2, Nothing))));` 0
- `countNodesByPredicate (fn (OneThing _) => true | _ => false) (OneThing (7, ManyThings ([1, 2], TwoThings (1, 2, OneThing (99, Nothing))));` 2
Part 8

countTwoThingsNodes: 'a ThingCollection \rightarrow \text{int}

Write the function countTwoThingsNodes that takes a 'a ThingCollection and returns the number of times the TwoThings constructor appears in the collection. You must define this function in terms of countNodesByPredicate. This means that your definition will consist of only a call to countNodesByPredicate; this function will not be recursive.

- countTwoThingsNodes Nothing; 0
- countTwoThingsNodes (OneThing (7, Nothing)); 0
- countTwoThingsNodes (OneThing (7, ManyThings ([1, 2], TwoThings (1, 2, Nothing)))); 1
- countTwoThingsNodes (OneThing (7, ManyThings ([1, 2], TwoThings (1, 2, OneThing (99, Nothing)))))}; 1

Part 9

reduceThingCollection: ('a * 'b \rightarrow 'b) \rightarrow 'b \rightarrow 'a ThingCollection \rightarrow 'b

Write the function reduceThingCollection that takes a function (f), a “base” value, and a 'a ThingCollection. reduceThingCollection function behaves much like the standard foldr function for lists. In short, when applied to the Nothing value, reduceThingCollection returns the “base” value; when applied to a OneThing (v, c) node, the result of f applied to the pair of v and the result of the recursive call is returned; when applied to a TwoThings (a, b, c) node, the result of f (a, f (b, result)) is returned, where result is the result of the recursive call; when applied to a ManyThings (L, c) node, the result is a foldr of L with the result of the recursive call as the base.

- val C = (OneThing (7, ManyThings ([4, 3], TwoThings (10, 8, OneThing (99, Nothing)))));
- reduceThingCollection (op +) 0 C; 131
- reduceThingCollection (op ::) [] C; [7, 4, 3, 10, 8, 99]
- reduceThingCollection Int.max 0 C; 99

Part 10

Consider the following datatype defining an n-ary tree holding integer values where each node may have any number of children (represented as a list of subtrees).

datatype NTree =
  NTreeNode of int * NTree list
| EmptyNTree

flattenNTree: NTree \rightarrow \text{int list}

Write the function flattenNTree that takes an NTree and returns the contents of the tree as a list where the value at each node precedes those of its children, in left-to-right order (i.e., a pre-order traversal).

- flattenNTree (NTreeNode (2, [NTreeNode (3, [EmptyNTree]), NTreeNode (4, []), NTreeNode (9, [])])); [2, 3, 4, 9]
- flattenNTree (NTreeNode (2, [NTreeNode (3, [EmptyNTree]),
  NTreeNode (4, []), NTreeNode (9, [NTreeNode (7, []), NTreeNode(6, [])]])); [2, 3, 4, 9, 7, 6]
You may use existing functions to simplify your implementation.

Logistics

- Strive for simplicity in your programming. Write short helper functions.
- Be certain that you can do each part of this assignment as you will use these features in later assignments. Ask lots of questions.
- Grading will be divided as follows.

<table>
<thead>
<tr>
<th>Part</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td>7</td>
<td>10</td>
</tr>
<tr>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>

- Get started now to avoid the last minute rush.