Assignment #2: ML Introduction #2

Note: Your ML code must be submitted in a file named hw2.sml. The Java code will be submitted in the classes specified in the problem statement.

Overview
This assignment is intended to continue your introduction to the ML programming language. Specifically, this assignment will explore algebraic data types, analogous representations in Java, and file I/O.

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 {value: 'a, next: 'a List}
    | EmptyList
```

Using this datatype, one can represent an int List analogous to the ML list [3,2,1] as follows.

```ml
ListNode {value=3, next=ListNode {value=2, next=ListNode {value=1, next=EmptyList}}}
```

lengthList: 'a List → int
Write the lengthList function to compute the length of a 'a List value. If invoked on the example above, the result would, as expected, be 3 (corresponding to the number of nodes).

Part 2
mapList: ('a → 'b) → 'a List → 'b List
Write the mapList function for the 'a List datatype. This function is analogous to the standard map for ML’s 'a list, but must be written in terms of the constructors above.

Part 3
“Stay awhile and listen.”
Let us pause for a moment.

Serious note: far too many people type a few things, make javac happy (even more do not bother to address all of the warnings), and then pass the test cases without really thinking about what is going on here and how it relates to the ML datatype definitions. Do not do that. I know time is short, you have others things to do, and you want to move on, but do not forget that this is a legitimate opportunity to learn.

It can take a while to get used to using these new datatype definitions, the corresponding constructors, and pattern matching. It can help to link these ideas to concepts with which you are already incredibly familiar and comfortable, as expressed in Java (if you have forgotten Java, that is ok, it will come back quickly enough for this part).

Once you complete this part, if the link between the familiar and the new becomes apparent, then the remaining parts should be significantly easier. And, in fact, you might come to appreciate just how awesome these language features are (they are objectively awesome, even if this is not how one would typically write Java code; that is not the claim here).

In the given files you will find multiple Java source files. Examine List.java, ListNode.java, and EmptyList.java (you can ignore all the equals and hashCode silliness). Notice how these correspond to the datatype (Java interface) given in Part 1 and its two constructors (Java classes). The data stored within a constructor is analogous to data stored within an object. This is an incredibly important point; one of the most common struggles during this course is forgetting to “unwrap” the data (i.e., trying to treat an object as though it was the data within; the types are all wonky).
Open Part3.java and read through the listLength implementation. Notice that this chaining of instanceof checks (and paired casts, where appropriate) accomplishes the same task as pattern-matching in ML.

To do: In Part3.java, complete the implementation of the map method in a similar manner as in Part 2 above.

You can run the provided unit tests to verify that the method behaves as expected. You might also examine the list creation in the unit tests to convince yourself that though the ML syntax can be quite verbose, it certainly isn’t more so than that of Java. Instead, we are just writing code in a different style than you may be accustomed to.

NOTE: Submit your files for this part as soon as you have completed it so that you do not forget. You need only submit the Part3.java file.

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

```plaintext
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 defining a representation of arithmetic expressions with variables (we will certainly revisit this in more detail as the course progresses).

```plaintext
datatype expression =
  ID of string
| NUM of int
| PLUS of expression * expression
| TIMES of expression * expression
;

gatherIdentifiers: expression -> string list
Write the function gatherIdentifiers that takes an expression and returns a list containing all of (the strings contained within) the identifiers within the expression (these should be in left-to-right order as they appear in the expression; duplicates are allowed).
```

Part 6
compute: expression -> int
Write a function named compute that takes an expression and returns the integer value that results from “computing” (what we will call evaluating) the expression. Input expressions might, of course, include identifiers as constructed by ID; for this assignment we will treat all identifiers as having integer value 2 (we will revisit identifiers in great detail later).

Part 7
simplifyIdentities: expression -> expression
Write the function simplifyIdentities that takes and expression and returns an expression. This function must simplify the input expression according to both the additive and multiplicative identities on integers (i.e., \( n + 0 = 0 + n = n \) and \( n \times 1 = 1 \times n = n \)), returning the simplified expression.

Tips: Pattern matching is the way to go here (and beyond). Simplify the subexpressions first and then pattern match against the resulting values.

Part 8
foldConstants: expression -> expression
Write a function named `foldConstants` that takes an expression and returns a new expression. The new expression is equivalent to the input expression but with all arithmetic between constant number values reduced to the result of the arithmetic. For instance, `PLUS (ID "a", PLUS (NUM 2, NUM 3))` (representing `a + (2 + 3)`) will fold to `PLUS (ID "a", NUM 5).

You are not expected to restructure the tree (based on associative and commutative properties) to expose additional opportunities to fold constants.

Part 9

Consider the following (revised) datatype defining a representation of expressions.

```plaintext
datatype expression2 =
  INT of int |
  ADD of expression2 * expression2 |
  MULT of expression2 * expression2 |
  LESSTHAN of expression2 * expression2
;
```

```plaintext
mixed_compute: expression2 → value
```

Write a function named `mixed_compute` that takes an expression and returns the result of “computing” (evaluating) the expression.

The evaluation of a `LESSTHAN` expression should result in a boolean value, whereas the evaluation of the other constructors should result in an integer value. A function can return only a single type of value, so you will need to define a new datatype (call it `value`) with constructors `INT_VAL`, `BOOL_VAL`, and `ERROR_VAL`.

If a result cannot be determined (e.g., evaluation leads to addition with a boolean value), then the result should be `ERROR_VAL`.

Part 10

```plaintext
wordsFromFile: string → string list
```

Write a function named `wordsFromFile` that takes a filename as a string and returns a list containing the “words” that appear in the file.

For this function, a “word” is any sequence of non-whitespace characters. You can use `Char.isSpace` to check for whitespace characters.

Since the purpose of this function to help prepare you for later assignments, you are not permitted to use `Strings.tokens` or any other similar function that does the majority of the work for you. You are, however, permitted (even encouraged) to use functions that you wrote for the first assignment.

You should also consider the use of (some of) the following library functions: `TextIO.openIn`, `TextIO.inputAll`, `TextIO.input1`, and `TextIO.endOfStream`. 
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.