Assignment #1: ML Introduction #1

Note: Your solution must be submitted in a file named `hw1.sml`.

Overview

This assignment is the first of a pair of assignments intended to serve as an introduction to the ML programming language. These assignments are not meant to cover the entire language, but should prepare you for the remaining programming assignments.

Note: Though each part asks that you “write a function”, you are always allowed to write additional helper/utility functions to simplify the implementation of the required function. This is good design in most languages, and is extremely important in this language where you will often write recursive solutions.

Mutation

Your solutions may not use mutation. This means that your solution cannot have variables of reference type, nor can it use data structures provided by the ML libraries that use mutation. This will be the case for all assignments in this course unless explicitly stated otherwise.

The purpose of this exercise is for you to familiarize yourself with the language, not to program in Java using the ML syntax.

Part of the programming experience in this course will, for most, be working in a new style. While you could write code in ML using a style more similar to the imperative background with which you are already familiar, doing so would be somewhat unnatural for the language and, more importantly, in so doing you would cheat yourself of an opportunity to think differently about programming and to develop new and very useful skills.

Part 1

**intToString: int → string**

Write a function named `intToString` that takes an integer and returns the string representation of the integer. You may use the built-in `Int.toString` function, but for negative integers your solution must return strings with ‘-’ instead of ‘∼’ (used by ML) as the negation operator.

For example,
- `intToString 99;`
  val it = "99" : string
- `intToString ~1;`
  val it = "-1" : string

Part 2

**twin: 'a list → ('a * 'a) list**

Write a function named `twin` that takes a list of values and returns a list of pairs. The returned list duplicates each element of the input list into a pair.

For example,
- `twin [1,2,3];`
  [(1,1),(2,2),(3,3)]

Part 3

**memberRec: "a list * "a → bool**
**memberExists: "a list * "a → bool**

You may ignore the inevitable “calling polyEqual” warning.

For this problem you will write the same function in two forms. This function implements a list membership test. Given a pair of a list of values and a singular value, the function must return `true` if the value appears in the list and `false` otherwise.

- Write a function named `memberRec` that implements this membership test. This function must be written using explicit recursion (i.e., as a traditional recursive function).
Write a function named `memberExists` that implements this membership test. This function must be written using a single call to `List.exists` (familiarity with higher-order functions is a valuable skill).

For example,
- `memberRec ([1,2,3], 2);`
  
  val it = true : bool
- `memberRec ([1,2,3], 4);`
  
  val it = false : bool
- `memberExists ([1,2,3], 2);`
  
  val it = true : bool
- `memberExists ([1,2,3], 4);`
  
  val it = false : bool

**Part 4**

`both`: (`'a → bool`) * (`'a → bool`) → `'a → bool`

Write a function named `both` that takes a pair of predicates and a value, `v`, of any type, and that returns `true` if `v` satisfies both predicates.

For example,
- `both (fn x => x > 0, fn x => x < 20) 10;`
  
  val it = true : bool
- `both (fn x => x > 0, fn x => x < 20) 100;`
  
  val it = false : bool

**Part 5**

`satisfiesAll`: (`'a → bool`) list * `'a → bool`

Write a function named `satisfiesAll` that takes a list of predicates and a value, `v`, of any type, and that returns `true` if `v` satisfies every predicate in the list.

For example,
- `satisfiesAll ([], 10);`
  
  val it = true : bool
- `satisfiesAll ([fn x => x > 0, fn x => x < 20], 10);`
  
  val it = true : bool
- `satisfiesAll ([fn x => x > 0, fn x => x < 20], 100);`
  
  val it = false : bool
- `satisfiesAll ([fn x => x > 0, fn x => x < 20, fn x => x mod 2 = 0], 10);`
  
  val it = true : bool
- `satisfiesAll ([fn x => x > 0, fn x => x < 20, fn x => x mod 2 = 0], 11);`
  
  val it = false : bool

**Part 6**

`mapSome`: (`'a → 'b option`) → `'a list` → `'b list`

Write a function named `mapSome` that takes a function, `f`, and a list. The `mapSome` function returns a list of values extracted from the SOME options returned by `f` applied to the elements of the input list. In the case that `f` applied to an element results in NONE, that element does not generate a corresponding value in the result list.

**Note:** no credit will be earned for solutions that use `List.mapPartial` since you are being asked to implement the same function.

For example,
- `mapSome (fn x => if x > 0 then SOME x else NONE) [1, 2, 3, 4, 55];`
  
  val it = [1,2,4] : int list
- `mapSome (fn x => if x > 0 then SOME (Int.toString x) else NONE) [1, 2, 3, 4, 55];`
  
  val it = ["1","2","4"] : string list
Part 7

prefixBy: ('a → bool) * 'a list → 'a list * 'a list

Write a function named prefixBy that takes, as a tuple, a predicate and a list. This function must return a pair of lists representing a split of the original list into a prefix and a suffix (i.e., the concatenation of the resulting lists will match the original list). This split is based on the provided predicate; all values that satisfy the predicate, prior to the first failure, must be placed in the first result list and all remaining values are placed in the second result list.

For example,
- prefixBy (fn n => n < 10, [1,4,5,9,10,3,4,21]);
  val it = ([1,4,5,9],[10,3,4,21])
- prefixBy (fn c => Char.isAlpha c, explode "abc123def");
  val it = ([#"a",#"b",#"c"],[#"1",#"2",#"3",#"d",#"e",#"f"])
- prefixBy (fn n => n < 10, [1,4,5,9]);
  val it = ([1,4,5,9],[1]) : int list * int list
- prefixBy (fn n => n < 10, [10,4,5,9]);
  val it = ([10,4,5,9]) : int list * int list

Part 8

groupAscending: int list → int list list

Write a function named groupAscending that takes a list of integers as an argument and returns a list of integer lists as its result. This function must group ascending (contiguous) sequences of integers in the original list into sublists in the result. More specifically, each sublist in the result represents an ascending sequence in the input list with the elements in the same relative order and the sublists in the same relative order (i.e., for two consecutive sublists, the elements of the second immediately followed the elements of the first in the original list).

For example,
- groupAscending [];
  val it = [] : int list list
- groupAscending [1];
  val it = [[1]] : int list list
- groupAscending [1,2,3];
  val it = [[1,2,3]] : int list list
- groupAscending [1,2,3,2,3];
  val it = [[1,2,3],[2,3]] : int list list
- groupAscending [4,7,10,2,3,1,99,45,122,123,122,47,46,46,49];
  val it = [[4,7,10],[2,3],[1,99],[45,122,123],[122],[47],[46],[46,49]] : int list list

Part 9

unescapeString: string → string

It is very common in programming languages to specify special characters in strings using escape sequences (e.g., the ubiquitous 
). These escape sequences, before being processed, are actually just two characters: a backslash (\) followed by another character completing the sequence.

For this problem, you are being asked to write a function to reduce each valid escape sequence into its corresponding single character. To highlight that such escape sequences are simply standardized conventions, and to avoid the common confusion that arises from testing escape sequences introduced with \ in a language that itself uses \ in its escape sequences, we will use : to introduce escape sequences.

For our purposes, the following are the only valid escape sequences.
Write a function named `unescapeString` that takes a string as its only argument. This function must return a string with the same characters as the argument string but with all valid escape sequences reduced to their corresponding single character complements. If an invalid escape sequence is found (i.e., a : character followed by something other than \n, \t, ", or :), then the function must raise an `InvalidEscapeSequence` exception with the offending character as an argument; if there is no character immediately following a :, then an `InvalidEscapeMissing` exception must be raised. (Copy the exception definitions given below into your source file.)

```ocaml
exception InvalidEscapeSequence of char;
exception InvalidEscapeMissing;
```

Note that you can convert between strings and lists with `explode` and `implode`.

For example,
```ocaml
- unescapeString "abc";
val it = "abc" : string
- unescapeString "ab :t c :\\n :: :n";
val it = "ab \t c \n" : \n" : string
- unescapeString "ab:bc" handle InvalidEscapeSequence c =>
  "invalid escape :" ^ str c;
val it = "invalid escape :b" : string
- unescapeString "ab:" handle InvalidEscapeMissing =>
  "escape sequence missing character"
val it = "escape sequence missing character" : string
```

Part 10

`escapeString: string → string`

Write a function named `escapeString` that takes a string as its only argument. This function acts as a complement to the `unescapeString` function from the previous part. More specifically, `escapeString` returns a string with the same characters as its argument string but with each of the “special” characters escaped in the result as outlined below.

<table>
<thead>
<tr>
<th>Found Character</th>
<th>Resulting Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>\n</td>
<td>:n</td>
</tr>
<tr>
<td>\t</td>
<td>:t</td>
</tr>
<tr>
<td>&quot;</td>
<td>:&quot;</td>
</tr>
<tr>
<td>::</td>
<td>::</td>
</tr>
</tbody>
</table>

For example,
```ocaml
- escapeString "abc";
val it = "abc" : string
- escapeString "ab \t c \n" : \n";
val it = "ab :t c :\\n :: :n" : string
```
Logistics

- Strive for simplicity in your programming. Write short helper functions.

- 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>