CSE 341, Autumn 2010
Assignment #1 - ML: Simple Functions (100 points)
Due: Thursday, October 7, 2010, 11:30 PM

In this assignment you will practice the ML building blocks described in pages 1-64 of the Ullman text by writing several short individual functions. Collect your answers for all questions in a single file called `hw1.sml`. There will be a turnin link available from the class web page. You are allowed to use the standard functions `abs` and `length`, but otherwise you are not allowed to use features that aren't described in pages 1-64 of the book. (If you like, you may use pattern matching and `let` expressions as taught during the second week of class.) Each function is worth up to 10 points.

You are expected to use good programming style, such as naming, indentation, and avoiding redundancy when possible. You should also place a descriptive comment on each of your functions.

If the solution to a previous problem is useful in helping you solve a later problem, you may call the earlier function from the later function. You are also allowed to define "helper" functions to solve these problems and you can include any testing data or code you develop (although it should be labeled as such and declared below the functions). If you want to use `let` constructs (which will be taught soon in lecture) to declare your helper functions, that is okay.

Several of these problems mention a *precondition* for a function. Such preconditions should always be commented. When a precondition is violated in ML, the right thing to do (just as it is in Java) is to throw an exception (it's called *raising* an exception in ML). If you want to work on developing good habits by including code to raise an exception when a precondition fails, that would be fine (exceptions are described starting on page 134 of the Ullman text). But we will ignore that code for grading because we aren't testing those cases.

Functions to Implement:

1. Define a function called `daysInMonth` that takes an integer representing a month as an argument and that returns the number of days in that month. You may assume that the month value passed is between 1 and 12 inclusive. You may also assume that the month is not part of a leap year. The following table shows the number of days in each month:

Month	1 Jan	2 Feb	3 Mar	4 Apr	5 May	6 Jun	7 Jul	8 Aug	9 Sep	10 Oct	11 Nov	12 Dec
Days	31	28	31	30	31	30	31	31	30	31	30	31

```
val daysInMonth = fn : int -> int
```

2. Define a function called `pow` that takes two integers as arguments and that returns the result of raising the first integer to the power of the second (i.e., `pow(x, y)` should return x^y). You may assume that the power is not negative. For our purposes, we will assume that every integer to the 0 power is 1 (this isn't true of 0 to the 0, but that's okay).

```
val pow = fn : int * int -> int
```

3. Define a function called `sumTo` that accepts an integer *n* and that computes the sum of the first *n* reciprocals. That is:

$$\sum_{i=1}^{n} \frac{1}{i}$$

For example, `sumTo(3)` should return (1 + 1/2 + 1/3) = 1.83333333.... The function should return 0.0 if *n* is 0. You may assume that the function is not passed a negative value of *n*.

```
val sumTo = fn : int -> real
```

4. Define a function called `repeat` that takes a string and an integer as arguments and that returns a string composed of the given number of occurrences of the string. For example, `repeat("hello", 3)` returns `"hellohellohello"`. You may assume that the function is not passed a negative value for the second argument.

```
val repeat = fn : string * int -> string
```

5. Define a function called `twos` that takes a single integer argument and that returns the number of factors of two in the number. For example, the number 168 when expressed as a product of prime factors is 2 * 2 * 2 * 3 * 7, which means that it has three factors of 2, so `twos(168)` should return `3`. Similarly the number -30 would be expressed as -(2 * 3 * 5), which means that it has one factor of 2. You may assume that the value passed to the function is not equal to 0.

```
val twos = fn : int -> int
```

6. Define a function called `absoluteDay` that takes two integers representing a month and day as parameters and returns the absolute day of the year between 1 and 365 that is represented by that month/day. For example, Jan 1 is absolute day #1; Jan 2 is #2; Jan 31 is #31; Feb 1 is #32; and so on, up to Dec 31, which is #365. You may assume that the month value passed is between 1 and 12 inclusive and that the day value passed is a positive integer. You may also assume that the date is not part of a leap year.

   HINT: If you were computing an absolute day in a procedural language like Java, you might use a cumulative sum loop to add up the days-in-month for all prior months, plus the days within the current month. For example:

	Jan		*Feb*		*Mar*		*Apr*		*May*		*Jun*		*Jul*		*total*
absolute day for July 24 =	31	+	28	+	31	+	30	+	31	+	30	+	24	=	**205**

   ```
   val absoluteDay = fn : int * int -> int
   ```

7. Define a function called `countNegative` that takes a list of integers as an argument and that returns a count of the number of negative integers in the list. For example, `countNegative([3,17,~9,34,~7,2])` should return `2`.

   ```
   val countNegative = fn : int list -> int
   ```

8. Define a function called `absList` that takes a list of `int * int` tuples and that returns a new list of `int * int` tuples where every integer is replaced by its absolute value. For example, `absList([(~38,47), (983,~14), (~17,~92), (0,34)])` should return `[(38,47), (983,14), (17,92), (0,34)]`.

   HINT: This is easier to solve if you write a helper function to process one tuple.

   ```
   val absList = fn : (int * int) list -> (int * int) list
   ```

9. Define a function called `split` that takes a list of integers as an argument and that returns a list of the tuples obtained by splitting each integer in the list. Each integer should be split into a pair of integers whose sum equals the integer and which are each half of the original. For odd numbers, the second value should be one higher than the first. For example, `split([5,6,8,17,93,0])` should return `[(2,3), (3,3), (4,4), (8,9), (46,47), (0,0)]`. You may assume that all of the integers in the list passed to the function are greater than or equal to 0.

   ```
   val split = fn : int list -> (int * int) list
   ```

10. Define a function called `isSorted` that takes a list of integers and that returns whether or not the list is in sorted (nondecreasing) order (`true` if it is, `false` if it is not). By definition, the empty list and a list of one element are considered to be sorted.

    ```
    val isSorted = fn : int list -> bool
    ```

11. Define a function called `collapse` that takes a list of integers as an argument and that returns the list obtained by collapsing successive pairs in the original list by replacing each pair with its sum. For example, `collapse([1,3,5,19,7,4])` should return `[4,24,11]` because the first pair (1 and 3) is collapsed into its sum (4), the second pair (5 and 19) is collapsed into its sum (24) and the third pair (7 and 4) is collapsed into its sum (11). If the list has an odd length, the final number in the list is not collapsed. For example, `collapse([1,2,3,4,5])` should return `[3,7,5]`.

    ```
    val collapse = fn : int list -> int list
    ```

12. Define a function called `insert` that takes an integer and a sorted (nondecreasing) integer list as parameters and that returns the list obtained by inserting the integer into the list so as to preserve sorted order. For example, `insert(8, [1,3,7,9,22,38])` should return `[1,3,7,8,9,22,38]`.

    ```
    val insert = fn : int * int list -> int list
    ```

13. Define a function called `insertionSort` that takes an integer list as a parameter and that returns the list obtained by sorting the list into nondecreasing order. This will be an ML solution to the classic insertion sort algorithm.

    HINT: In writing `insertionSort`, it is useful to have a helper function that can insert a single value into its proper place in a sorted sub-list of the overall list. Where might you find such a function?...

    If you are unfamiliar with insertion sort, see Wikipedia: http://en.wikipedia.org/wiki/Insertion_sort

    ```
    val isort = fn : int list -> int list
    ```

Extra Credit (+1 point): factors

As will always be the case, extra credit problems will be worth very few points, especially when you consider how much work they involve. Extra credit is for people who want to explore a little further. Remember that you are limited to constructs in pages 1—64 of the Ullman text.

14. *(1 point)* Define a function called `factors` that takes an integer as an argument and that returns an ordered list of the factors of that integer. Recall that a factor is a number that goes evenly into another. For example, the call `factors(12)` should return `[1,2,3,4,6,12]`. Your list has to include the factors in increasing order without duplicates. You may assume that that the value passed to the function is greater than 0. You will need to write a helper function to solve this task and it will take more than one argument. Your solution is required to run in $O(n^{1/2})$ time (time proportional to the square root of n).

    HINT: This does not require a lot of code. Our solution is a one-line function that calls a 5-line helper function.

    ```
    val factors = fn : int -> int list
    ```

Grading and Submission:

For reference, our solution is 48 lines long (54 with the extra credit), excluding blank lines and comments. You don't need to match this number exactly or even come close to it; it's just a rough guideline.

Your program should not produce any syntax errors or warning messages when loaded into the ML interpreter, such as "non-exhaustive match." You may lose points if you name your functions incorrectly, even if their behavior is correct.

Your code should work for both basic and advanced cases. Perform your own testing, and remember to test edge cases such as negative numbers, empty lists, zeros, etc. unless the problem says you can assume that you won't encounter those.

You are expected to use good programming style, such as naming, indentation/spacing, and avoiding redundancy when possible. Avoid long lines of over 100 characters in length; if you have such a line, split it into multiple lines.

Favor the `::` operator to grow lists versus the `@` operator (though the `@` operator is not forbidden entirely).

Place a descriptive comment heading at the top of your program along with a header on each function, including any preconditions. Redundancy, such as recomputing a value unnecessarily or unneeded recursive cases, should be avoided.