Assignment 2
Due Monday November 10 at 1pm.
No late assignments will be accepted.

The questions below require you to write ML functions. Some of the problems in this assignment require a mix of functional and non-functional programming, and specifically, the use of references, assignment statements and iteration. However, unless a question explicitly requires the use of such imperative programming features, your programs should be purely functional. Pattern matching should also be used whenever possible, and in general, simple solutions are preferred and will receive the most marks. Feel free to use helper functions wherever appropriate. Unless otherwise specified, you may assume that the input to your functions is correct, so that no error checking is required. Finally, by properly raising exceptions, your functions in this assignment should not produce any warnings of the form match nonexhaustive.

You should hand in four files: the source code of all your ML functions, a sample terminal session with the ML interpreter, the answers to pencil-and-paper problems, and a signed and completed cover sheet. The source code should be well commented, and the terminal session should be short and should demonstrate that your functions work correctly. These files should be submitted electronically using the submission web page.

Note: The marker has a limited amount of time for each assignment, so it is your responsibility to provide documentation and testing that allows him to quickly evaluate your work. As with all work in this course, 20% of the grade is for quality of presentation.

No more questions will be added

1. **Basic Recursion and Pattern Matching** (10 points total)

   Using recursion, define an ML function `linear(A,B,X)` of type `real*real*(real list) -> (real list)`. If \( x_i \) is the \( i^{th} \) element of list \( X \), then \( A + Bx_i \) is the \( i^{th} \) element of the output list. For example,

   \[
   \text{linear}(1.0, 2.0, [3.1, 4.2, 5.7]) \\
   \Rightarrow [1.0+2.0*3.1, 1.0+2.0*4.2, 1.0+2.0*5.7] \\
   \Rightarrow [7.2, 9.4, 12.4]
   \]
Do not use any map functions in your solution. Define the function in two ways: (a) without pattern matching (5 points), and (b) with pattern matching (5 points). (These two versions of the function should be given different names, such as linear1 and linear2.)

2. Record Types and Exceptions (33 points total)

(a) (3 points)
Define flight to be a named type for records, where each record has these fields: number, airline, from, to, duration, passengers, where number is an integer, airline, from and to are strings (e.g., “air canada”, “toronto”, “vancouver”, respectively), duration is real, and passengers is a list of strings (e.g., [“john”, “paul”, “george”, “ringo”]).

(b) (8 points)
Define an ML function updateD(D,F) of type real*flight -> flight that changes the duration of flight F to D, i.e., the function returns a copy of F with the duration changed. Raise an exception if D is not positive.

(c) (10 points)
Define an ML function removePassenger(P,F) of type string*flight -> flight that removes passenger P from flight F. If P is not in the passenger list of flight F, then raise an exception.

(d) (12 points)
Define an ML function fastest(A,B,L) of type string*string*(flight list) -> real*int that returns the duration and flight number of the flight in L that takes the least time to go from A to B. If there are no such flights, then raise an exception.

3. Exception Handling (20 points)

This question builds on the previous one. Define an ML function updatePassengers(L1,L2) where L1 has type (string*int*string) list, and L2 has type flight list, and the output has the same type as the input, (L1,L2). For each tuple (P,N,A) in L1, the function should search L2 for a flight with number N and airline A, and then remove passenger P from this flight using the function removePassenger that you defined in Question 2(c). If no such flight exists, then add the tuple (P,N,A) to a list of “exceptional” tuples. Likewise, if removePassenger raises an exception, then updatePassengers should catch the exception, leave the flight unchanged and add the tuple (P,N,A) to the list of “exceptional” tuples. Finally, updatePassengers should return the list of exceptional tuples and the list of updated flights. Note that some flights may not be updated, and some may be updated multiple times. In all cases, however, the length of the output flight list should be the same as the length of the input flight list (since flights are not added or deleted from the list).

updatePassengers should be purely functional and should not use any references or assignment statements, so the updated flight list that is returned should be a copy of L2 with all the successful changes made. You may assume that L2 contains no more than one flight for each pair of values (N,A).
4. **Variant Types** (15 points total).

(a) (5 points) Define a variant datatype called \texttt{length} that allows real numbers to represent meters, yards, feet, inches and centimeters.

(b) (5 points) Define a function \texttt{convert} of type \texttt{length -> real} that converts any length into inches.

(c) (5 points) Define a function \texttt{totalLength} of type \texttt{length list -> real} that takes a list of lengths and returns the sum of the lengths in inches.

5. **Recursive Types** (40 points total).

(a) (5 points) Define a recursive datatype for representing trees in which each internal node can be either red or black. Red nodes have two children, and black nodes have three children. All red nodes store a positive integer, and all black nodes store a negative integer. A leaf can store a value of any type.

(b) (4 points) Draw a tree and show how it is represented in ML using your datatype. The tree should have three levels (including the root) and at least one node of each colour. What is the type of this tree?

(c) (13 points) Define a function \texttt{largestRed(T)} that returns the red node in tree \texttt{T} with the largest integer. Raise an exception if there are no red nodes.

(d) (10 points) Define a function \texttt{addTree(N,T)} that adds \texttt{N} to each integer in the internal nodes of tree \texttt{T}. Raise an exception if this results in a node getting an illegal value (i.e., a red node getting a non-positive value, or a black node getting a non-negative value).

(e) (8 points) Define a function \texttt{mapTree(F,T)} that applies function \texttt{F} to each of the values in the leaves of tree \texttt{T}, and returns a list of the results. Test \texttt{mapTree} using both named and unnamed functions for \texttt{F}.

6. **References and Iteration** (30 points total)

This question asks you to implement the same program in three different ways: as a purely functional program, as a purely procedural program, and as a mixed functional/procedural program.

(a) (8 points) \textbf{Functional.} Suppose \texttt{L} is a list of real numbers. Define a function \texttt{updateList1(X,N,L)} that returns a copy of \texttt{L} with \texttt{X} added to the \texttt{N}th element. If \texttt{L} has fewer than \texttt{N} elements, then raise an exception. The function should be purely functional (and thus have no side effects). Thus, \texttt{updateList1} has type \texttt{real*int*(real list) -> real list}. For example,

\[
\text{updateList1}(5.2, 3, \{6.1,7.1,8.1,9.1\}) \Rightarrow \{6.1,7.1,13.3,9.1\}
\]

(b) (10 points) \textbf{Functional/Procedural.} Suppose that \texttt{L} is a list of references to real numbers. Using recursion, define a function \texttt{updateList2(X,N,L)} that adds \texttt{X} to the \texttt{N}th number referenced in \texttt{L} . If \texttt{L} has fewer than \texttt{N} elements, then raise an exception. Unlike \texttt{updateList1}, \texttt{updateList2} is not purely functional but has side effects. That is, it does not return a new list; instead, it returns the
unit element, (), and changes the existing list, L. Thus, **updateList2** has type **real*int**(real ref list) -> unit. Here is a sample session with the ML interpreter:

- val L = [ref 6.1, ref 7.1, ref 8.1, ref 9.1];
  val L = [ref 6.1, ref 7.1, ref 8.1, ref 9.1]

- updateList2(5.2,3,L);
  val it = ()

- L;
  val it = [ref 6.1, ref 7.1, ref 13.3, ref 9.1]

- updateList2(2.4,2,L);
  val it = ()

- L;
  val it = [ref 6.1, ref 9.5, ref 13.3, ref 9.1]

(c) (12 points) **Procedural.**
Define a function **updateList3** of type **real*int**(real ref list) -> unit that behaves just like **updateList2**, but which is defined using while loops instead of recursion.

7. **Hash Tables** (65 points total).

In Question 2, you defined a record type called **flight**. In this question, you will build a hash table for storing and accessing flight records. A hash table is an array in which each array position is the start of a chain of records (i.e., a linked list). The chain at position \( p \) of the array stores all records that hash to position \( p \). In this question, assume that a flight record hashes to position \( p = n \mod m \) where \( n \) is the flight number and \( m \) is the number of positions in the hash table. Also assume that each chain is sorted in descending order by flight number (so that the highest flight number is first in a chain).

Note that different airlines may use the same flight number. Thus there may be records in a hash table for both “Air Canada flight 107” and “American Airlines flight 107”. However, each record is uniquely identified by its flight number and airline. That is, a hash table should not contain two records with the same flight number and airline.

In the questions below, you should define functions that efficiently maintain all these requirements. In particular, except for **createTable**, each function should use at most a constant amount of new storage. To do this, you will have to use iteration instead of recursion, and you will have to make extensive use of references in your chains. Do not use a sorting routine to maintain the chains in sorted order.

(a) (5 marks)
Define a type or datatype called **hashTable** for hash tables storing flight records.
Define a function `createTable(M)`, of type `int->hashTable`, that creates and returns an empty hash table with $M$ positions.

Define a function `insertFlight(F,H)`, of type `flight*hashTable->unit`, that inserts a flight record, $F$, into hash table $H$. Raise an exception if a record with the same flight number and airline is already present in $H$.

Define a function `removeFlight(N,A,H)`, of type `int*string*hashTable->unit`, that removes the record with flight number $N$ and airline $A$ from hash table $H$. Raise an exception if such a record does not exist.

Define a function `getFlight(N,A,H)`, of type `int*string*hashTable->flight`, that retrieves the record with flight number $N$ and airline $A$ from hash table $H$. Raise an exception if the record is not there.

Define a function `printChain(C,H)`, of type `int*hashTable->unit`, that prints the flight number and airline of every record in chain $C$ of hash table $H$ in the order in which they are stored in the chain. (This function will help the grader to determine if your hash table has the right structure.)

No more questions will be added
Complete this page and submit it with your assignment.

Name: ________________________________  
(Underline your last name)

Student number: ________________________________

I declare that this assignment is solely my own work, and is in accordance with the University of Toronto Code of Behaviour on Academic Matters.

Signature: ________________________________