

## **Assignment 2B**

**Due Friday March 20** at 11:50am, just before tutorial.

No late assignments will be accepted.

---

The questions below require you to write ML functions. Unless stated otherwise, (i) programs should be purely functional whenever possible, (ii) pattern matching should be used whenever possible, and (iii) simple programs are preferred to complex ones. Feel free to use helper functions wherever appropriate. You should hand in two files: a source listing of all your ML functions, and a transcript of a terminal session with the ML interpreter. These files should be submitted electronically using the submission web page. The source code should be well commented, and the terminal session should be short but should demonstrate that your functions work correctly. To keep things simple, you may assume that the input to your functions is correct, so that no error checking is required.

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.

---

## 1. Function Composition (11 marks total).

In the following questions, you should define a small collection of unary functions to test your functions with.

- (a) (1 point) Define a function `triple(F)` that composes the unary function `F` with itself three times. Thus,

```
(triple t1) [1,2,3,4,5,6] => (t1 (t1 (t1 [1,2,3,4,5,6])))  
                        = [4,5,6]
```

```
(triple square) 2 => (square (square (square 2))) = 256
```

- (b) (5 points) Define a function `compose(F,N)` that composes a unary function `F` with itself `N` times. If `N` is negative, then raise an exception. Thus,

```
compose(square,4) => (fn X => (square (square (square (square X)))))
```

```
compose(t1,4) [1,2,3,4,5,6,7] => (t1 (t1 (t1 (t1 [1,2,3,4,5,6,7])))  
                                = [5,6,7]
```

```
compose(square,~2) => raise exception
```

- (c) (5 points) Define a function `composeAll(L)` that composes a list `L` of unary functions. Thus,

```
composeAll [t1,t1,t1,t1] => (fn X => (t1 (t1 (t1 (t1 X)))))
```

```
composeAll [sub1,square,add1] => (fn X => (sub1 (square (add1 X))))
```

```
(composeAll [f,g,h]) x => (f (g (h x)))
```

## 2. Exceptions (11 points total).

- (a) (3 points) Define a function `average(L)` that returns the average of a list `L` of integers. If `L` is empty, then an exception should be raised.
- (b) (3 points) Define a function `average2(L)` that returns a pair `(p,n)`, where `p` is the average of the positive numbers in `L` and `n` is the average of the negative numbers in `L`. If there are no positive numbers in `L`, then an exception should be raised. If there are no negative numbers in `L`, then a different exception should be raised.
- (c) (5 points) Suppose that `L` has the form `[L1,L2,...,Ln]`, where each `Li` is a list of integers. Define a function `sumAvg(L)` that returns

```
(average L1) + (average L2) + ... + (average Ln)
```

Your function should use the `average` function defined in part (a) as a helper function. In particular, if a call to `average` raises an exception, then `sumAvg` should handle it by pretending that the average is 1. Thus,

`sumAvg [[1,2,3], [4,6], []] => 2.0 + 5.0 + 1.0 = 8.0`

### 3. Traversing Trees (16 points total).

- (a) (2 points) Define a datatype, `btree`, for trees in which each node stores an integer and can have 0, 1 or 2 children.
- (b) (3 points) Define a function `sum0(T)` that returns the sum of the integers stored in nodes with 0 children in tree `T`.
- (c) (3 points) Define a function `sum1(T)` that returns the sum of the integers stored in nodes with 1 child in tree `T`.
- (d) (3 points) Define a function `sum2(T)` that returns the sum of the integers stored in nodes with 2 children in tree `T`.
- (e) (5 points) Define a function `sumAll(T)` that returns a triple `(i,j,k)` as output, where `i = sum0(T)`, `j = sum1(T)` and `k = sum2(T)`. However, your function should traverse tree `T` only once, so it cannot use `sum1`, `sum2` or `sum3` as helper functions.

### 4. Updating Lists (7 points total).

- (a) (2 points) Suppose `L` is a list of integers. Define a function `double1(L)` that returns a copy of `L` with all its elements doubled. The function should be purely functional (and thus have no side effects). Thus,

`double [1,2,3,4,5] => [2,4,6,8,10]`

- (b) (5 points) Suppose that `L` is a list of integer references. Define a function `double2(L)` that replaces each integer referenced in `L` by twice its value. Unlike `double1`, `double2` should have side effects: it does not return a new list; instead, it returns the unit element, `()`, and it changes the existing list, `L`. For example, here is a sample terminal session:

```
- val L = [ref 1, ref 2, ref 3, ref 4];
  val L = [ref 1,ref 2,ref 3,ref 4]

- double2 L;
  val it = ()

- L;
  val it = [ref 2,ref 4,ref 6,ref 8]

- double2 L;
  val it = ()

- L;
  val it = [ref 4,ref 8,ref 12,ref 16]
```

## 5. Updating Binary Search Trees (30 points total).

Recall that a binary search tree (BST) is a tree in which each node stores a key (and possibly other data) and has at most two children. The keys are unique in that each key appears at most once in the tree. In addition, each node,  $N$ , of the tree must satisfy the following *BST property*: the key at node  $N$  is greater than all keys in the left subtree of  $N$  and less than all keys in the right subtree. In the following questions, your ML programs should exploit and enforce the BST property (by using the standard BST algorithms that you learned in earlier courses). Any changes to a BST should be done using assignment statements. When problems or unexpected conditions arise, exceptions should be raised. You should define various types of exception, one for each type of problem described below. If several problems occur at once, you need only raise an exception for one of them.

- (a) (3 points) Define a datatype for binary search trees for bank accounts. Each node should store a record with two fields: `account_number` and `balance`. The first field stores an integer and is the key for the BST. The second field stores a reference to a real number.
- (b) (3 points) Define an ML function `assets(RT)` that returns the sum of all the account balances stored in a BST. Here, `RT` is a reference to the BST.
- (c) (5 points) Define an ML function `addInterest(P,RT)` that adds  $P$  percent interest to every bank account in a BST. Here, `RT` is a reference to the BST, and `P` is an integer (where, for example, `P=15` represents 15%). The function should raise an exception if `P` is negative. As in Question 4, the function should return the unit tuple, `()`.
- (d) (5 points) Define an ML function `deposit(Amt,Acct,RT)` that adds an amount to the balance of an account in a BST. Here, `Amt` is the amount, `Acct` is the account number, and `RT` is a reference to the BST. The function should raise an exception if the amount is negative, and it should raise a different exception if the account is not in the BST. It should return the unit tuple, `()`.
- (e) (5 points) Define an ML function `setBalance(Bal,Acct,RT)` that sets the balance of an account in a BST, overwriting the old balance with a new balance. Here, `Bal` is the new balance, `Acct` is the account number, and `RT` is a reference to the BST. The function should raise an exception if the new balance is negative, and it should raise a different exception if the account is not in the BST. It should return the unit tuple, `()`.
- (f) (9 points) Define an ML function `newAccount(Acct,RT)` that creates a new account with balance 0 and adds it to the BST. Here, `RT` is a reference to the BST, and `Acct` is the number of the new account. The function should raise an exception if an account with that number already exists in the BST. It should return the unit tuple, `()`.

## 6. Transaction Processing (12 points total).

This question builds on the previous one. Your main task is to define a function that accepts a list of transactions on bank accounts, executes them, and prints error messages when exceptions are raised. The accounts are stored in a binary search tree (BST), as in the previous question.

- (a) (2 points) Define a datatype for the transactions. The following transaction types should be included:
- i. **Assets**: print the sum total of all account balances.
  - ii. **Int(P)**: add P percent interest to every account.
  - iii. **Dep(Amt,Acct)**: deposit amount **Amt** to account **Acct**.
  - iv. **Set(Bal,Acct)**: set the balance of account **Acct** to **Bal**.
  - v. **New(Acct)**: create a new account with account number **Acct** and balance 0.
- (b) (3 points) Define a function `execute(Trans,RT)` that executes a transaction on a BST. Here, **Trans** is the transaction, and **RT** is a reference to the BST. This function should use the functions in Question 5 *without modification*, and it should *not* catch any exceptions. It should return the unit tuple, `()`.
- (c) (7 points) Define a function `process(List,RT)` that processes a list of transactions on a BST, catches exceptions, and prints error messages. Here, **List** is the list of transactions, and **RT** is a reference to the BST. The function `execute` defined above should be used to execute individual transactions. If an exception is raised during the execution of a transaction, then `process` should catch the exception, print an error message, and go on to execute the remaining transactions. An error message should describe the problem and identify the transaction in which it occurred. A transaction should be identified by its position in **List**. (*e.g.*, the third transaction in **List** is called transaction 3.) There should be a different error message for each type of problem. For example,

Error: attempt by transaction 17 to deposit into a non-existing account.

Error: attempt by transaction 45 to set the balance of a non-existing account.

Error: attempt by transaction 97 to create an already existing account.

Error: attempt by transaction 106 to deposit a negative amount.

Error: attempt by transaction 245 to set a negative balance.

SUGGESTION: In some questions in this assignment, you may find it helpful to use *case statements*. These have the following form:

```
case <expr> of
  <pattern1> => <expr1>
| <pattern2> => <expr2>
  ...
| <patternN> => <exprN>
```

Case statements behave very much like pattern matching in function definitions. First, `<expr>` is evaluated. If its value matches `<pattern1>`, then `<expr1>` is evaluated and returned. Otherwise, if its value matches `<pattern2>`, then `<expr2>` is evaluated and returned. Etc. Case statements are particularly useful for pattern matching against references. For example,

```
case !R of
  1 => R:=4
| 2 => R:=7
| 3 => R:=5
| N => R:=2*N
```

Here, `R` is a reference to an integer, and the case statement updates the reference depending on the value of the integer. In particular, it doubles the value of most integers, but gives special treatment to the integers 1, 2 and 3, which are changed to 4, 7 and 5, respectively.

## Cover sheet for Assignment 2B

---

Complete this page and attach it to the front of 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 Behavior on Academic Matters.

**Signature:** \_\_\_\_\_