University of Toronto Mississauga
CSC 324 - Principles of Programming Languages, Fall 2016

Assignment 2

Due Tuesday November 15 at 11pm.

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 and should use recursion. Pattern matching should also be used whenever possible, unless it complicates the code significantly. 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 non-exhaustive*. You may have to detect illegal inputs in order to avoid such warnings, and in such cases, you should print out an error message.

Unless specified otherwise, do not use any built-in functions that would require recursion if you defined them yourself. You may, of course, use any function you like if you define it yourself (in terms of allowed functions). The point here is that you should not scour the user manual or the web for functions that will solve most of a problem for you. You may use the append operator, `@`.

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** (20 points total)

   Using recursion, define an ML function `listSum(A,X,Y)` of type `real*(real list)*(real list) -> (real list)`. If x_i is the i^{th} element of list `X`, and y_i is the i^{th} element of list `Y`, then $A + x_i * y_i$ is the i^{th} element of the output list. For example,

```
listSum(7.3, [3.1, 4.2, 5.7], [2.7, 4.1, 1.5])
      => [7.3 + 3.1 * 2.7,  7.3 + 4.2 * 4.1,  7.3 + 5.7 * 1.5]
      => [15.67, 24.52, 15.85]
```

   If lists `X` and `Y` do not have the same length, then raise an exception. Your function should traverse the lists only once. Do not use any `map` functions in your solution. Define the function in two ways: (a) without pattern matching (10 points), and (b) with pattern matching (10 points). These two versions of the function should be called `listSum1` and `listSum2`, respectively.

2. **Record Types and Exceptions** (56 points total)

   (a) (3 points)
   Using `type`, define `student` to be a named type for student recorda, where each record has three fields: `id`, `name` and `gpa`, of type `int`, `string` and `real`, respectively. For example, {`id=1234, name=''Spock'', gpa=97.1`} is a record of type `student`. This record means that the student with `id`=1234 has name Spock and a gpa of 97.1.

   (b) (3 points)
   Likewise, define `taken` to be a record with three fields: `course`, `student` and `grade`, of type `string`, `int` and `real`, respectively. For example, {`course=''csc324'', student=1234, grade=87.2`} is a record of type `taken`. This record means that the student with `id`=1234 has taken course `csc324` and received a grade of 87.2.

   (c) (5 points)
   Define an ML function `updateGPA(G,S)` of type `real*student -> student` that changes the gpa of student `S` to `G`. i.e., the function returns a copy of `S` with the gpa field changed. Raise an exception if `G` is negative.

   In the questions below, `Slist` is list of `student` records, and `Tlist` is list of `taken` records. For each value of `id`, `Slist` should contain only one `student` record. Likewise, for each pair of values for `course` and `student`, `Tlist` should contain only one `taken` record. Other than this, the lists are arbitrary.

   (d) (10 points)
   Define an ML function `updateGrade(T,Tlist)` of type `taken*(taken list) -> (taken list)`. This function updates the grade a student received in a given course. Specifically, if `T` = {`course=C, student=N, grade=G`}, then the function searches `Tlist` for a record with `course=C` and `student=N`. It then changes the grade in this record to `G`. That is, the function returns a copy of `Tlist` in which the grade has been changed in the appropriate

record. If `Tlist` does not contain such a record, then an exception should be raised.

(e) (20 points)
Define an ML function `names(C,Slist,Tlist)` of type `string*(student list)*(taken list) -> (string list)`. This function returns the names of all students who have taken course `C`, according to the data in `Slist` and `Tlist`. Raise an exception if a student who has taken the course is not listed in `Slist`.

(f) (15 points)
Define an ML function `computeGPA(I,Tlist)` of type `int*(taken list) -> real`. This function computes the gpa of the student with `id=I` from the courses he has taken, as given in `Tlist`. If the student has not taken any courses, then an exception is raised.

3. **Exception Handling** (20 points)
This question builds on the previous one. Define an ML function `updateAllGPAs(Slist,Tlist)` of type `(student list)*(taken list) -> (student list)*(student list)`. This function uses `computeGPA` from the previous question to compute the gpa of each student in `Slist`. If `computeGPA` raises an exception, then `updateAllGPAs` catches and handles it by putting the student on a list of exceptional students. Otherwise, `updateAllGPAs` uses `updateGPA` to produce an updated student record containing the computed gpa, and puts this record onto a list of updated student records. Finally, `updateAllGPAs` returns both the list of updated student records and the list of exceptional records, in that order. Every student in the input list should appear in one of the two output lists. `updateAllGPAs` should be purely functional and should not use any references or assignment statements.

4. **Variant Types** (30 points total)
We would like to extend the integers to include infinity and negative infinity. We also want to extend arithmetic operators to them. For example, we would like $1/0 = infinity$, $1 + infinity = infinity$, $-4 * infinity = -infinity$, $3/infinity = 0$, $infinity * infinity = infinity$, etc. Unfortunately, this cannot always be done. For example, the values of $0/0$, $infinity - infinity$, $infinity/infinity$ and $0 * infinity$ are all undefined. In such cases, your functions below should raise an exception. Of course, your functions should reduce to ordinary integer arithmetic when the arguments are ordinary (finite) integers.

Note that if you define a datatype for extended integers naively, then functions defining extended integer arithmetic will have to handle many cases involving positives and negatives. For instance, $infinity/3 = infinity$, $infinity/(-3) = -infinity$, $-infinity/3 = -infinity$, $-infinity/(-3) = infinity$. You should define your datatype for extended integers in a way that reduces the number of cases that your functions must consider. Your functions should also make maximal use of pattern matching.

(a) (5 points) Define a variant datatype called `eInt` for extended integers.

(b) (5 points) Define a function called `eAdd` of type `eInt*eInt -> eInt` that performs addition on extended integers.

(c) (5 points) Define a function called `eSub` of type `eInt*eInt -> eInt` that performs subtraction on extended integers.

(d) (5 points) Define a function called `eMult` of type `eInt*eInt -> eInt` that performs multiplication on extended integers.

(e) (5 points) Define a function called `eDiv` of type `eInt*eInt -> eInt` that performs division on extended integers. (Recall that `div`, not `/`, is the ML operator for integer division.)

(f) (5 points) Use your functions to evaluate the arithmetic expressions below. That is, write each expression in terms of `eAdd`, `eMult`, `eSub` nd `eDiv`, and then evaluate them using ML.

i. $13 - (6/0)$
ii. $3 * (1/0)$
iii. $(-2/0) * (-6/0)$
iv. $(5/0) + (-3/0)$
v. $(7/0)/(3/(2-2))$
vi. $3/(-4/(5/0))$
vii. $3/(-4/(5/(-6/0)))$
viii. $4 + (7 * (9/(8-5)))$
ix. $(4 - (3+1)) * (3/(4 + (-4)))$
x. $(4-4)/(6-(3+3))$

5. **Recursive Types** (50 points total).
Your functions in this question should traverse a tree at most once.

(a) (5 points) Define an ML datatype called `'a tree` for representing trees. The internal nodes of a tree store a value of any type and may have 1, 2 or 3 children. The leaves of a tree may store a real number or a list of strings.

(b) (5 points) Draw a tree and show how it is represented in ML using your datatype. The tree should have at least two levels of internal nodes, at least one node with one child, at least one with two children and at least one with three children. It should also have at least one leaf storing a real number and at least one storing a list of strings.

(c) (10 points) Define a function `list12(T)` of type `'a tree -> 'a list` that returns of list of all the values stored at nodes having 1 or 2 children in tree `T`.

(d) (10 points) Define a function `countNodes(T)` of type `'a tree -> int*int*int` that returns a tuple `(N1,N2,N3)` where `N1` is the number of nodes in tree `T` having 1 child, `N2` is the number having 2 children, and `N3` is the number having 3 children.

(e) (10 points) Define a function `treeApply(F,T)` of type `(real->real)*('a tree) -> real` that applies function `F` to every real number stored in the leaves of tree `T` and sums the results.

(f) (10 points) define a function `leafAppend(L,T)` of type `(string list)*('a tree) -> ('a tree)` that appends list `L` to the front of every list stored in the leaves of tree `T`. `leafAppend` should be purely functional and should not actually modify `T`, but should return a modified copy.

6. **References and Iteration** (45 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. Each of your programs should traverse a list at most once. In parts (b) and (c), you will have to be careful in your use of brackets. For example, `!f(x)` should be written as `!(f(x))`, for otherwise it will be parsed as `(!f)(x)`.

(a) (10 points) *Functional.*
We shall use the following datatype to represent lists of integers:

```
datatype list1 = nil1 | cons1 of int*list1
```

For example, the expression `cons1(4,cons1(5,cons1(6,nil1)))` represents the list `[4,5,6]`. Define a function `remList1(N,L)` of type `int*list1 -> list1` that returns a copy of `L` with the N^{th} element removed. If `L` has fewer than `N` elements, then raise an exception `NoSuchElement`. The function should be purely functional (and have no side effects). For example, if

```
L => cons1(4,cons1(5,cons1(6,nil1)))
```

then here is a sequence of ML expressions and their values:

```
remlist1(1,L) => cons1(5,cons1(6,nil1))
L => cons1(4,cons1(5,cons1(6,nil1)))
remlist1(2,L) => cons1(4,cons1(6,nil1))
L => cons1(4,cons1(5,cons1(6,nil1)))
remlist1(3,L) => cons1(4,cons1(5,nil1))
L => cons1(4,cons1(5,cons1(6,nil1)))
remlist1(4,L) => exception NoSuchElement
L => cons1(4,cons1(5,cons1(6,nil1)))
```

Notice that the value of `L` does *not* not change.

(b) (15 points) *Functional/Procedural.*
We shall now use the following datatype to represent lists of integers:

```
datatype list2 = nil2 | cons2 of int*(list2 ref)
```

For example, the list `[4,5,6]` is represented by the expression

```
cons2(4,ref(cons2(5,ref(cons2(6,ref(nil2))))))
```

Using recursion (not iteration), define a function `remList2(N,RL)` of type `int*(list2 ref) -> unit` that removes the N^{th} element from the list referenced by `RL`. If the list has fewer than `N` elements, then raise an exception `NoSuchElement`. Unlike `remList1`, `remList2` is not purely functional but has side effects. That is, it does not return a new list; instead, it changes the existing list. For example, if

```
RL => ref(cons2(4,ref(cons2(5,ref(cons2(6,ref(nil2)))))))
```

then here is a sequence of ML expressions and their values:

```
remList2(2,RL) => ()
RL => ref(cons2(4,ref(cons2(6,ref(nil2)))))
remList2(2,RL) => ()
RL => ref(cons2(4,ref(nil2)))
remList2(2,RL) => exception NoSuchElement
RL => ref(cons2(4,ref(nil2)))
remList2(1,RL) => ()
RL => ref(nil2)
remList2(1,RL) => exception NoSuchElement
RL => ref(nil2)
```

Notice that the value of `RL` *does* change.

Hint: Define a function `cdr2(L)` of type `list2 -> (list2 ref)`. If `L => cons(N,RL)`, then `cdr2(L) => RL`. Otherwise, if `L => nil2`, then `cdr2(L)` raises an exception.

(c) (20 points) *Procedural.*
Define a function `remList3(N,RL)` of type `int*(list2 ref) -> unit` that behaves just like `remList2`, but which is defined using while loops instead of recursion.

No more questions will be added

University of Toronto Mississauga
CSC 324 - Principles of Programming Languages

Cover sheet for Assignment 2

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: _____