Prog. Lang. Design and Implementation	Homework 1
ENGR 3599 Spring 2014 (Pucella)	Due Date: Friday, Jan 31

This homework is meant to be done individually. You may discuss problems with fellow students, but all work must be entirely your own, and should not be from any other course, present, past, or future. If you use a solution from another source, please cite it. If you consult fellow students, please indicate their names.

Submission information:

- For this problem set, solutions should be put in a file called `homework1.sml`.
- Details of submission to follow in the coming days, once I figure them out. Irrespective of the submission details, solutions will be due at the beginning of class on the due date.
- Your file `homework1.sml` should begin with a block comment that lists your name, your email address, and any remarks that you wish to make about the homework.

Notes:

- All code should compile in the latest release version of Standard ML of New Jersey.
- There is a file `homework1.sml` available from the web site containing code you can use, including code described in this write-up.
- All questions are compulsory.

Question 1 (Warmup)

(a) Code a function `gcd` with type

```
int -> int -> int
```

where `gcd m n` computes the greatest common divisor of integers `m` and `n`. Note that while `m` and `n` may be negative, the greatest common divisor is always taken to be positive.

```
- gcd 0 10;
val it = 10 : int
- gcd 2 10;
val it = 2 : int
- gcd 3 10;
val it = 1 : int
- gcd 4 10;
val it = 2 : int
- gcd 5 10;
val it = 5 : int
- gcd ~5 10;
val it = 5 : int
```

(b) Code a function `lcm` with type

```
int -> int -> int
```

where `lcm m n` computes the least common multiple of integers `m` and `n`. Note that while `m` and `n` may be negative, the least common multiple is always taken to be positive.

```
- lcm 0 10;
val it = 0 : int
- lcm 2 10;
val it = 10 : int
- lcm 3 10;
val it = 30 : int
- lcm 4 10;
val it = 20 : int
- lcm 5 10;
val it = 10 : int
- lcm ~5 10;
val it = 10 : int
```

(c) Code a function `exp` with type

```
int -> int -> int
```

where `exp a n` computes a^n (exponentiation), defined by $a^0 = 1$ and $a^{n+1} = a\,a^n$. If `n` is less than zero, treat it as if it were zero.

```
- exp 0 2;
val it = 0 : int
- exp 2 2;
val it = 4 : int
- exp 2 3;
val it = 8 : int
- exp 2 4;
val it = 16 : int
- exp 3 4;
val it = 81 : int
- exp 4 4;
val it = 256 : int
```

(d) Code a function `tetra` with type

```
int -> int -> int
```

where `tetra a n` computes ${}^n a$ (tetration), defined by ${}^0 a = 1$, and ${}^{n+1} a = a^{({}^n a)}$. If `n` is less than zero, treat it as if it were zero.

```
- tetra 0 2;
val it = 1 : int
- tetra 1 2;
val it = 1 : int
- tetra 2 1;
val it = 2 : int
- tetra 2 2;
val it = 4 : int
- tetra 2 3;
val it = 16 : int
- tetra 2 4;
val it = 65536 : int
- tetra 3 2;
val it = 27 : int
- tetra 4 2;
val it = 256 : int
- tetra 5 2;
val it = 3125 : int
```

Question 2 (Lists and Matrices)

(a) Code a function `sum` with type

```
int list -> int
```

where `sum xs` returns the sum of all the integers in list `xs`.

```
- sum [];
val it = 0 : int
- sum [1];
val it = 1 : int
- sum [1,2];
val it = 3 : int
- sum [1,2,3];
val it = 6 : int
- sum [1,2,3,4];
val it = 10 : int
```

(b) Code a function `prod` with type

```
int list -> int
```

where `prod xs` returns the product of all the integers in list `xs`.

```
- prod [];
val it = 1 : int
- prod [1];
val it = 1 : int
- prod [1,2];
val it = 2 : int
- prod [1,2,3];
val it = 6 : int
- prod [1,2,3,4];
val it = 24 : int
```

(c) Code a function `every_other` with type

```
'a list -> 'a list
```

where `every_other xs` returns a list made up of every other element of `xs`: If `xs` is `[`a_1`,` a_2`,` a_3`, ...]`, then the result is `[`a_1`,` a_3`,` a_5`, ...]`

```
- every_other [];
stdIn:105.1-105.15 Warning: type vars not generalized because of
   value restriction are instantiated to dummy types (X1,X2,...)
val it = [] : ?.X1 list
- every_other [1];
val it = [1] : int list
- every_other [1,2];
val it = [1] : int list
- every_other [1,2,3];
val it = [1,3] : int list
- every_other [1,2,3,4];
val it = [1,3] : int list
- every_other [1,2,3,4,5];
val it = [1,3,5] : int list
- every_other [1,2,3,4,5,6];
val it = [1,3,5] : int list
```

(Don't worry about the warning in `every_other []`.)

(d) Code a function `flatten` with type

```
'a list list -> 'a list
```

where `flatten xss` takes a list of lists `xs` and returns the list made up of appending together all the lists in `xs` in the order in which they appear in `xs`.

```
- flatten [];
stdIn:115.1-115.11 Warning: type vars not generalized because of
   value restriction are instantiated to dummy types (X1,X2,...)
val it = [] : ?.X1 list
- flatten [[]];
stdIn:116.1-116.13 Warning: type vars not generalized because of
   value restriction are instantiated to dummy types (X1,X2,...)
val it = [] : ?.X1 list
- flatten [[1,2],[3,4,5]];
val it = [1,2,3,4,5] : int list
- flatten [[1,2],[],[3,4,5]];
val it = [1,2,3,4,5] : int list
```

(e) Code a function `heads` with type

```
'a list list -> 'a list
```

where `heads xss` takes a list of lists `xs` and returns the list made up of all the first elements of the lists in `xs`, skipping the empty lists.

```
- heads [];
stdIn:119.1-119.9 Warning: type vars not generalized because of
   value restriction are instantiated to dummy types (X1,X2,...)
val it = [] : ?.X1 list
- heads [[]];
stdIn:120.1-120.11 Warning: type vars not generalized because of
   value restriction are instantiated to dummy types (X1,X2,...)
val it = [] : ?.X1 list
- heads [[1,2],[3,4,5]];
val it = [1,3] : int list
- heads [[1,2],[],[3,4,5]];
val it = [1,3] : int list
```

(f) Code a function `tails` with type

```
'a list list -> 'a list list
```

where `tails xss` takes a list of lists `xs` and returns the list made up of all the tails of the lists in `xs`, skipping the empty lists. (The tail of a list `ys` is the list made up of all but the first element of `ys`.)

```
- tails [];
stdIn:124.1-124.9 Warning: type vars not generalized because of
   value restriction are instantiated to dummy types (X1,X2,...)
val it = [] : ?.X1 list list
- tails [[]];
stdIn:125.1-125.11 Warning: type vars not generalized because of
   value restriction are instantiated to dummy types (X1,X2,...)
val it = [] : ?.X1 list list
- tails [[1,2],[3,4,5]];
val it = [[2],[4,5]] : int list list
- tails [[1,2],[],[3,4,5]];
val it = [[2],[4,5]] : int list list
- tails [[1,2],[3],[4,5,6]];
val it = [[2],[],[5,6]] : int list list
```

(g) We can represent a matrix of integers as a list of lists, by listing all the rows in top to bottom order. That is, the matrix

$$\begin{pmatrix} a_1 & a_2 \\ a_3 & a_4 \end{pmatrix}$$

can be represented by the list `[[`a_1`, `a_2`], [`a_3`, `a_4`]]`.

Code a function `scaleMat` with type

```
int -> int list list -> int list list
```

where `scaleMat a m` takes an integer `a` and a matrix `m` as a list of rows, and returns the matrix obtained by scaling every element of `m` by `a`.

```
- scaleMat 0 [[1,2],[3,4]];
val it = [[0,0],[0,0]] : int list list
- scaleMat 1 [[1,2],[3,4]];
val it = [[1,2],[3,4]] : int list list
- scaleMat 2 [[1,2],[3,4]];
val it = [[2,4],[6,8]] : int list list
- scaleMat 3 [[1,2],[3,4]];
val it = [[3,6],[9,12]] : int list list
```

(h) Code a function `addMat` with type

```
int list list -> int list list -> int list list
```

where `addMat m1 m2` takes two matrices `m1` and `m2` (as lists of rows) and returns the matrix representing the sum of `m1` and `m2`. (Assume for simplicity the matrices are well formed, such as every row having the same size, and that `m1` and `m2` have the same size. If not, then do as we did in `addVec` and `inner`, and stop when one of the lists is empty. You can use `addVec` from the supplied code if it helps, of course.)[1]

```
- addMat [[1,2],[3,4]] [[1,2],[3,4]];
val it = [[2,4],[6,8]] : int list list
- addMat [[1,1],[1,1]] [[1,2],[3,4]];
val it = [[2,3],[4,5]] : int list list
- addMat [[1,2,3],[4,5,6]] [[10,20,30],[40,50,60]];
val it = [[11,22,33],[44,55,66]] : int list list
```

Bonus Questions: If you're really bored, try the following questions (these are optional):

- Code a function `transpose` with type

  ```
  'a list list -> 'a list list
  ```

  where `transpose m` returns the transpose of matrix `m`. (Again, you can assume that the matrix is well formed, and stop when one of the lists is empty.)

[1] I know this is sloppy. But I don't want us to get distracted by error-checking at this point in the game. A real implementation of matrices would check that matrices are well formed when they are constructed. We may do that later. Or not.

```
- transpose [[1,2],[3,4]];
val it = [[1,3],[2,4]] : int list list
- transpose [[1,2,3],[4,5,6]];
val it = [[1,4],[2,5],[3,6]] : int list list
```

- Code a function `mulVecMat` with type

  `int list -> int list list -> int list`

  where `mulVecMat v m` multiplies row vector `v` (as a list of integers) and matrix `m` (as a list of lists of integers), returning a row vector. (Assume well-formedness, and that the sizes match, stopping when lists are empty.)

```
- mulVecMat [1,1] [[1,2],[3,4]];
val it = [4,6] : int list
- mulVecMat [1,0] [[1,2],[3,4]];
val it = [1,2] : int list
- mulVecMat [0,1] [[1,2],[3,4]];
val it = [3,4] : int list
- mulVecMat [1,2] [[1,2],[3,4]];
val it = [7,10] : int list
```

- Code a function `mulMat` with type

  `int list list -> int list list -> int list list`

  where `mulMat m1 m2` multiplies matrix `m1` by matrix `m2`. (Assume well-formedness, and that the sizes match, stopping when lists are empty.)

```
- mulMat [[1,0],[0,1]] [[1,2],[3,4]];
val it = [[1,2],[3,4]] : int list list
- mulMat [[0,1],[1,0]] [[1,2],[3,4]];
val it = [[3,4],[1,2]] : int list list
- mulMat [[1,2],[3,4]] [[1,0],[0,1]];
val it = [[1,2],[3,4]] : int list list
- mulMat [[1,2],[3,4]] [[0,1],[1,0]];
val it = [[2,1],[4,3]] : int list list
- mulMat [[1,1],[1,1]] [[1,2],[3,4]];
val it = [[4,6],[4,6]] : int list list
```

Question 3 (Internal Representation with Matrices)

The code provided for this homework includes the implementation of the internal representation and interpreter for the simple mathematical language that I described in Lecture 02 (minus Booleans and conditionals): integers and vectors with addition, multiplication, subtraction, and negation as operations.

The internal representation has been augmented with support for matrices of integers:

```
datatype expr = EInt of int
              | EVec of int list
              | EMat of int list list    (* MATRIX EXPRESSION *)
              | EAdd of expr * expr
              | ESub of expr * expr
              | EMul of expr * expr
              | ENeg of expr
              | EDiv of expr * expr
```

(Don't worry about `EDiv`, it is for Question 4. You can ignore it for now.)

Addition, multiplication, negation, and subtraction make sense for matrices. In this question, we will implement the appropriate interpretation for those operations over matrices.

Since now interpretation can result in matrices, values have also been augmented with matrices:

```
datatype value = VInt of int
               | VVec of int list
               | VMat of int list list     (* MATRIX VALUE *)
               | VRat of int * int
```

(Again, don't worry about `VRat`, it is for Question 4. You can just ignore it for now.)

(a) Fix the implementation of `eval` so that it evaluates `EMat` correctly.

```
- eval (EMat [[1,2],[3,4]]);
val it = VMat [[1,2],[3,4]] : value
- eval (EMat [[1,2,3],[3,4,5]]);
val it = VMat [[1,2,3],[3,4,5]] : value
```

(b) Fix the implementation of the interpreter so that you can add matrices, negate matrices, subtract matrices, and multiply a matrix by an integer to scale it.

```
- eval (EAdd (EMat [[1,2],[3,4]], EMat [[10,20],[30,40]]));
val it = VMat [[11,22],[33,44]] : value
```

```
- eval (EAdd (EMat [[1,2,3],[4,5,6]], EMat [[10,20,30],[40,50,60]]));
val it = VMat [[11,22,33],[44,55,66]] : value
- eval (ENeg (EMat [[1,2],[3,4]]));
val it = VMat [[~1,~2],[~3,~4]] : value
- eval (EAdd (EMat [[1,2,3],[4,5,6]], ENeg (EMat [[10,20,30],[40,50,60]])
   ));
val it = VMat [[~9,~18,~27],[~36,~45,~54]] : value
- eval (ESub (EMat [[1,2,3],[4,5,6]], EMat [[10,20,30],[40,50,60]]));
val it = VMat [[~9,~18,~27],[~36,~45,~54]] : value
- eval (EMul (EInt 3, EMat [[1,2,3],[4,5,6]]));
val it = VMat [[3,6,9],[12,15,18]] : value
```

If you implemented matrix multiplication in Question 2, feel free to fix the implementation of the interpreter so that you can multiply matrices together, and multiply a row vector by a matrix.

To think about: In the internal representation we're currently using, vector and matrices are hardwired to take integer constants: we have `EVec [1,2,3]` and `EMat [[1,2],[3,4]]`. Suppose we changed the internal representation so that vectors (and matrices) took arbitrary expressions:

```
...
| EVec of expr list
| EMat of expr list list
...
```

You could then write `EVec [EPlus (EInt 1, EInt 2),EInt 4]`, which is more reasonable. (Later on, when we add variables, you would be able to use variables to construct vectors, something that you couldn't do with the internal representation we're using now.)

What change would you need to make to the interpreter to support this sort of internal representation? Can you make it work? (Don't submit it—experiment on a copy of the homework file so that you don't mess things up.) Try it out for vectors first, it's simpler. Note that vector values are still lists of integers. So it better be the case that the expressions used to construct vectors evaluate to integers.

It might require a little bit of trickery to get SML to accept the code, which is why I did not assign this problem for real. In particular, you're probably going to need mutually-recursive functions.

Question 4 (Internal Representation with Rationals)

In this question, we examine what we need to do to add support for division to the mathematical language.

First, the internal representation has been augmented with support for division, in the form of an expression `EDiv (`e_1`, `e_2`)`:

```
datatype expr = EInt of int
              | EVec of int list
              | EMat of int list list
              | EAdd of expr * expr
              | ESub of expr * expr
              | EMul of expr * expr
              | ENeg of expr
              | EDiv of expr * expr         (* DIVISION EXPRESSION *)
```

(Expression `EMat` was handled in Question 3.)

The problem with division, of course, is that dividing two integers need not yield an integer. Therefore, we are going to allow expressions to evaluate to rational numbers, and extend addition, subtraction, negation, and multiplication to rational numbers.

The first thing to do is to augment values with rational numbers:

```
datatype value = VInt of int
               | VVec of int list
               | VMat of int list list
               | VRat of int * int          (* RATIONAL NUMBER VALUE *)
```

(Values `VMat` were handled in Question 3.) A rational number is represented by a pair `(`n`,`d`)` of a numerator n and denominator d.

(a) Code a function `simplifyRat` with type:

```
int * int -> value
```

where `simplifyRat r` takes a rational number `r` as a pair of integers and returns a value representing that rational number in its most simplified form: both the numerator and denominator are simplified, and if the denominator happens to be 1, then the value produced should in fact be an integer. Give some thought to negative rational numbers as well. (If the denominator is zero, you should raise the `DivisionByZero` exception.)

```
- simplifyRat (1,2);
val it = VRat (1,2) : value
```

```
- simplifyRat (0,1);
val it = VInt 0 : value
- simplifyRat (1,0);

uncaught exception DivisionByZero
  raised at: solution1.sml:120.18-120.46
- simplifyRat (2,4);
val it = VRat (1,2) : value
- simplifyRat (4,2);
val it = VInt 2 : value
- simplifyRat (4,6);
val it = VRat (2,3) : value
- simplifyRat (6,4);
val it = VRat (3,2) : value
- simplifyRat (~1,~2);
val it = VRat (1,2) : value
- simplifyRat (1,~2);
val it = VRat (~1,2) : value
- simplifyRat (2,~1);
val it = VInt ~2 : value
```

(b) Code a function `addRat` with type:

```
int * int -> int * int -> value
```

where `addRat r s` takes two rational numbers and adds them together to produce a value in its most simplified form.

```
- addRat (1,2) (1,3);
val it = VRat (5,6) : value
- addRat (1,2) (2,3);
val it = VRat (7,6) : value
- addRat (1,2) (1,2);
val it = VInt 1 : value
- addRat (2,4) (4,2);
val it = VRat (5,2) : value
- addRat (2,4) (5,2);
val it = VInt 3 : value
```

(c) Code a function `mulRat` with type:

```
int * int -> int * int -> value
```

where `mulRat r s` takes two rational numbers and multiplies them together to produce a value in its most simplified form.

```
- mulRat (1,2) (1,2);
val it = VRat (1,4) : value
- mulRat (1,2) (2,1);
val it = VInt 1 : value
- mulRat (1,3) (2,1);
val it = VRat (2,3) : value
- mulRat (1,3) (1,4);
val it = VRat (1,12) : value
- mulRat (2,3) (4,2);
val it = VRat (4,3) : value
```

(d) Code a function `negRat` with type:

```
int * int -> value
```

where `negRat r` takes a rational number and negates it to produce a value in its most simplified form.

```
- negRat (0,1);
val it = VInt 0 : value
- negRat (1,2);
val it = VRat (~1,2) : value
- negRat (4,~3);
val it = VRat (4,3) : value
- negRat (~3,~2);
val it = VRat (~3,2) : value
```

(e) Fix the implementation of `eval` so that it evaluates `EDiv` correctly. Note that `EDiv` should work with both integers and rational numbers, in any combination. (So that nested `EDiv` can be interpreted correctly.)

```
- eval (EDiv (EInt 1, EInt 2));
val it = VRat (1,2) : value
- eval (EDiv (EInt 2, EInt 4));
val it = VRat (1,2) : value
- eval (EDiv (EInt 4, EInt 6));
val it = VRat (2,3) : value
- eval (EDiv (EInt 1, EDiv (EInt 1, EInt 2)));
val it = VInt 2 : value
- eval (EDiv (EDiv (EInt 1, EInt 2), EDiv (EInt 1, EInt 2)));
val it = VInt 1 : value
- eval (EDiv (EDiv (EInt 1, EInt 2), EInt 2));
val it = VRat (1,4) : value
```

(f) Fix the implementation of the interpreter so that you can add rational numbers, negate them, subtract them, and multiply them. Note that it makes perfect sense to add a rational number and an integer, or multiply a rational number by an integer, so all of those cases should be handled as well.

```
- eval (EAdd (EDiv (EInt 1, EInt 2), EDiv (EInt 1, EInt 3)));
val it = VRat (5,6) : value
- eval (EMul (EDiv (EInt 1, EInt 2), EDiv (EInt 1, EInt 3)));
val it = VRat (1,6) : value
- eval (EAdd (EDiv (EInt 1, EInt 2), EInt 3));
val it = VRat (7,2) : value
- eval (EMul (EInt 2, EDiv (EInt 1, EInt 2)));
val it = VInt 1 : value
```

To think about: Suppose you implemented vectors (similarly for matrices, but let's focus on vectors for now) over expressions as opposed to integers, as I asked you to think about at the end of Question 3. It might make sense then to have vectors of rational numbers, or even vectors with a mix of rational numbers and integers. What change would you need to make to `value` to get this to work? Once you have this, it makes perfect sense then to multiply a vector by a rational number, which is a well-defined scaling operation. What do you need to implement the scaling of a vector by a rational number? What about adding two vectors of rational numbers?