Homework 3 — Simple ML programs — assigned Tuesday 6 March — due Tuesday 20 March

3.1 Boolean formulae: writing recursive functions over algebraic datatypes (25pts) [A.1; C; K.1.1; K.2.3; K.2.4]

We can use the following declaration to introduce a language of Boolean formulae:

```
datatype expr = Const of bool
              | Var of int
              | And of expr list
              | Or of expr list
              | Not of expr
```

For instance, the Boolean formula $(\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2)$ is represented by the ML term `And [Or [Not (Var 1), Var 2, Var 3], Or [Var 1, Not (Var 2)]]`. The `And` and the `Or` lists have at least two elements (conjuncts/disjuncts).

3.1.1 Simple Boolean evaluator (5pts)

Write a function `eval`, with type `env -> expr -> bool`, which computes the Boolean value of a formula.

The type `env` is the type of environments; an environment is simply an assignment of Boolean values to variables x_i. Some concrete representation must be chosen for `env`, and we choose a binary tree, as follows:

```
datatype ('a,'b) searchtree =
   Empty
 | Node of 'a * 'b * ('a,'b) searchtree * ('a,'b) searchtree
fun lookup (equal: 'a * 'a -> bool) (lessthan: 'a * 'a -> bool)
           (t: ('a,'b) searchtree) (k: 'a)
  : 'b option =
  case t of
     Empty => NONE
   | Node (key,value,left,right) =>
        if equal (k,key) then
           SOME value
        else if lessthan (k,key) then
           lookup equal lessthan left k
        else
           lookup equal lessthan right k
type env = (int, bool) searchtree
```

3.1.2 More on Boolean formulae: satisfiability checker (10pts)

Write a function `satisfiable: expr -> bool`, which determines if the given formula is satisfiable, i.e., true for some assignment of Boolean values to the variables that appear in the formula. Note: efficiency is not a concern.

3.1.3 More on Boolean formulae: tautology checker (10pts)

Write a function `tautology: expr -> bool`, which determines if the given formula is a tautology, i.e., true for *all* possible assignments of Boolean values to the variables that appear in the formula. Note: efficiency is not a concern.

3.2 An arithmetic expression evaluator: writing recursive functions over algebraic datatypes (15pts) [C; K.1.1; K.2.3; K.2.4; K.2.7]

We use the following data type declaration to introduce a language of simple arithmetic expressions, with variables and binding:

```
datatype expr = Num of int
              | Var of string
              | Let of {var: string, value: expr, body: expr}
              | Add of expr * expr
              | Sub of expr * expr
              | Mul of expr * expr
              | Div of expr * expr
type env = string -> int
exception Unbound of string
val emptyEnv: env = fn s => raise (Unbound s)
fun extendEnv oldEnv s n s' = if s' = s then n else oldEnv s'
exception ExprDivByZero
```

Write a function `evalInEnv`, with type `env -> expr -> int`, which computes the arithmetic value of an expression (which may have free variables) in a given environment (a mapping from variables to `int` values).

Then define:

```
fun eval e = evalInEnv emptyEnv e
```

so that `eval` evaluates closed expressions.

3.3 Number representations: rationals (15pts) [A.3; E; K.2.2]

Use the following representation for rational numbers:

```
type rat = IntInf.int * IntInf.int
```

The representation invariants are as follows. An SML value `(p,q)` represents the rational number $\frac{p}{q}$. The denominator q is positive and $\frac{p}{q}$ is a reduced fraction.

Implement the following operations over rationals, with the obvious mathematical meaning:

1. (1pt) `toString: rat -> string`; e.g., `toString (3, 4)` should evaluate to `"3/4"`.
2. (2pts) `fromInt: int -> rat`, to make a rational from an ordinary SML `int`.
3. (12pts) `addRat: rat * rat -> rat`,
   `subRat: rat * rat -> rat`,
   `mulRat: rat * rat -> rat`,
   `divRat: rat * rat -> rat`,
   `ltRat: rat * rat -> bool`,
   and `eqRat: rat * rat -> bool`.

3.4 Polymorphic arithmetic: ordered fields and matrices over them (30pts) [A.2; A.3; E; K.2.2; K.2.3]

In this exercise you are asked to put together a software framework using components you worked on in previous homework assignments.

The following type declarations will be used:

```
type 'a lvector = 'a list
type 'a lmatrix = 'a list list
type 'a orderedfield =
   {
    tostring: 'a -> string,
    add: 'a * 'a -> 'a,
    mul: 'a * 'a -> 'a,
    addid: 'a,
    mulid: 'a,
    addinv: 'a -> 'a,
    mulinv: 'a -> 'a,
    lt: 'a * 'a -> bool,
    eq: 'a * 'a -> bool
   }
val realorderedfield: real orderedfield =
   {
    tostring = Real.toString,
    add = Real.+,
    mul = Real.*,
    addid = 0.0,
    mulid = 1.0,
```

```
    addinv = Real.~,
    mulinv = fn x => 1.0 / x,
    lt = Real.<,
    eq = Real.==
   }
```

Matrices have at least one row and at least one column, and are represented in row-major form (a matrix is a list of rows, and a row is a list of elements). All operations are assumed to have conformant arguments.

Define the following polymorphic functions for linear algebra:

3.4.1 (1pt)

`gmI: 'a orderedfield -> int -> 'a lmatrix`, to make the identity matrix of given size.

3.4.2 (1pt)

`gvsp: 'a orderedfield -> 'a lvector * 'a -> 'a lvector`, to multiply a vector by a scalar.

3.4.3 (1pt)

`gmsp: 'a orderedfield -> 'a lmatrix * 'a -> 'a lmatrix`, to multiply a matrix by a scalar.

3.4.4 (1pt)

`gvvs: 'a orderedfield -> 'a lvector * 'a lvector -> 'a lvector`, to add two vectors.

3.4.5 (1pt)

`gmms: 'a orderedfield -> 'a lmatrix * 'a lmatrix -> 'a lmatrix`, to add two matrices.

3.4.6 (1pt)

`gvvp: 'a orderedfield -> 'a lvector * 'a lvector -> 'a`, to compute the inner product of two vectors.

3.4.7 (4pts)

`gmt: 'a lmatrix -> 'a lmatrix`, to transpose a matrix.

3.4.8 (4pts)

`gmvp: 'a orderedfield -> 'a lmatrix * 'a lvector -> 'a lvector`, to compute a matrix-vector product.

3.4.9 (5pts)

`gmmp: 'a orderedfield -> 'a lmatrix * 'a lmatrix -> 'a lmatrix`, to compute a matrix-matrix product.

3.4.10 (5pts)

`gminv: 'a orderedfield -> 'a lmatrix -> 'a lmatrix`, to invert a matrix.

3.4.11 (5pts)

`gmdet: 'a orderedfield -> 'a lmatrix -> 'a`, to compute the determinant of a matrix.

3.4.12 (1pts)

Using the type `rat` from the preceding exercise, declare a value `ratorderedfield: rat orderedfield`, to collect the various operations on rationals.

Thoroughly test your polymorphic functions for linear algebra using these two examples:

- The built-in reals, `realorderedfield`.
- The rationals, `ratorderedfield`.

Implementations should be reasonably time- and space-efficient. For instance, computing the determinant by cofactor expansion takes more than polynomial time and is not considered efficient.

3.5 Higher-order functions [C; E; K.1.1; K.2.1; K.2.2; K.2.3; K.2.4] (15pts)

We declare a data type of trees where each branch node may have any finite number of branches, as follows:

```
datatype t = L of int
           | N of t list
val example = N [L 1, N [L 2, N [L 3, L 4], L 5], N [L 6, L 7], L 8]
```

Given this datatype declaration, we could declare a value `size`, which is a function to count the leaves, as follows:

```
fun size (L n) = 1
  | size (N l) = List.foldr Int.+ 0 (List.map size l)
```

Instead, we introduce a combinator function `K` and a bottom-up fold `tfold` over the type `t`, such that we can declare `size` as follows:

```
val size = tfold {fL = K 1, fN = List.foldr Int.+ 0}
```

Write a declaration that declares `K` and `tfold` such that in its scope the above (second) declaration for `size` is valid (has a type) and correct (always computes the same result as the first declaration and the verbal specification).

How to turn in

Submission instructions: see course mailing list.

Include the following statement with your submission, signed and dated:
I pledge my honor that in the preparation of this assignment I have complied with the University of New Mexico Board of Regents' Policy Manual.