Homework 3 — ML core language — assigned Tuesday 7 March — due Tuesday 21 March

3.1 Boolean formulae: writing recursive functions over algebraic datatypes (15pts) [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 (5pts)

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 (5pts)

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 Using lists for arithmetic: writing recursive functions over lists (20pts) [A.3; E; K.2.2]

This is a continuation of homework exercise 2.1, and all conventions carry over from that exercise. Write the following functions for computing with arbitrarily large numbers (nonnegative integers) represented as lists of digits in arbitrary radices:

1. (2pts) `subLongInts: numeral * numeral -> numeral`, such that
   `subLongInts` (a,b)
   computes the difference $a - b$. Precondition is that $a \geq b$.
2. (1pt) `eqLongInts: numeral * numeral -> bool`, such that
   `eqLongInts` (a,b)
   determines if a and b are equal.
3. (2pts) `ltLongInts: numeral * numeral -> bool`, such that
   `ltLongInts` (a,b)
   determines if $a < b$.
4. (15pts) `divmodLongInts: numeral * numeral -> numeral * numeral`, such that
   `divmodLongInts` (a,b)
   computes the quotient and the remainder in the division a/b. Precondition is that b is not 0.

3.3 Drawing: writing recursive functions over lists; manipulating strings (15pts) [A.4; E; K.2.2]

In this exercise, we develop some simple tools for drawing.

A drawing is just a line drawing consisting of some number of polygons. A polygon is given as a list of vertices, and a vertex is simply a pair of real numbers for the x and y coordinates.

For instance,

```
[[(100.0,100.0),(100.0,200.0),(200.0,100.0)],
[(150.0,150.0),(150.0,200.0),(200.0,200.0),(200.0,150.0)]]
```

is an internal representation in ML of a drawing consisting of a triangle and a square.

3.3.1 Creating the drawing command (10pts)

Your task is to convert such a representation of a drawing into a simple page description in the PostScript language. Specifically, you are to write an ML function `makeCommand: (real * real) list list -> string`.

The result returned by `makeCommand` is an ML value of type string, which must contain valid PostScript commands for drawing the given polygons. When the SML/NJ top level prints this result, print it using `print`, and paste it by hand into a file *result.ps*, and then display it by running GhostScript: `gs result.ps`.

For instance, the expression

```
makeCommand [[(100.0,100.0),(100.0,200.0),(200.0,100.0)],
[(150.0,150.0),(150.0,200.0),(200.0,200.0),(200.0,150.0)]]
```

should evaluate to the string:

```
%!PS-Adobe-3.0 EPSF-3.0
%%BoundingBox: 100.0 100.0 200.0 200.0

100.0 100.0 moveto
100.0 200.0 lineto
200.0 100.0 lineto
closepath
stroke

150.0 150.0 moveto
150.0 200.0 lineto
200.0 200.0 lineto
200.0 150.0 lineto
closepath
stroke

showpage
%%EOF
```

which will be displayed as in Figure 1.

Note that the bounding box is the smallest upright rectangle such that no points of the drawing lie outside it; it is specified by giving the coordinates of its lower left and upper right corners, in our example $(100.0, 100.0)$ and $(200.0, 200.0)$.

The example shown here entirely suffices as a pattern to follow; however, if you would like to learn more about the PostScript language you can follow the links on the course web page.

Hint: you need to learn how to control the top level of SML/NJ so that it does not truncate very long strings when it prints them.

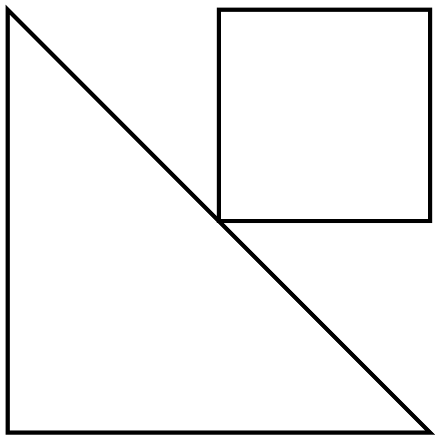

Figure 1: A triangle and a square.

3.3.2 Streams (file input/output) (5pts)

Read about file output in ML in Paulson, Section 8.8, and in the documentation of the Standard ML Basis Library. Produce the output PostScript file directly from ML instead of cutting and pasting. (Implement a function `dumpStringToFile: string * string -> unit`, which takes two strings; the first is the name of the file to be written, and the second is the string that is to be dumped to the file.)

3.4 Using lists for text (10pts) [K.2.2]

Write a function `split_into_words` to split text into words. Spaces, tabs, and new lines are word separators.

The function you write will replace the comment in the code fragment below to make the whole work.

```
local
   fun acceptfile (fileName: string) : string =
      let
         val f = TextIO.openIn fileName
         val s = TextIO.inputAll f
         val _ = TextIO.closeIn f
      in
         s
      end
   type word = char list
   type sentence = char list
   (* your code goes here *)
   fun split_into_words (s: sentence): word list =
   (* begin your code *)

   (* end your code *)
in
```

```
   fun main (filename: string) =
      let
         val s = acceptfile filename
         val cl = explode s
         val wl = split_into_words cl
      in
         wl
      end
end
```

The declaration of `split_into_words` should be expressed in terms of the language primitives (i.e., without the use of library functions).

3.5 An arithmetic expression evaluator: writing recursive functions over algebraic datatypes (10pts) [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.

3.6 Number representations: signed integers (15pts) [A.3; E; K.2.2]

Use the following representation for signed integers:

```
datatype sign = Pos | Neg
datatype numeral = Numeral of {sign: sign, magnitude: int}
```

The representation invariants are as follows. Positive signed integers are represented as a `Pos` sign and a positive `magnitude`; negative signed integers are represented as a `Neg` sign and a positive `magnitude`; zero is represented as a `Pos` sign and 0 `magnitude`.

Implement the following operations over signed integers:

1. (2pts) `toString: numeral -> string`, which produces the common mathematical representation of a number, with an explicit sign; `fromInt: int -> numeral`, to make a signed integer from an ordinary SML `int`; and `toInt: numeral -> int`, to make an ordinary SML `int` from a signed integer.
2. (10pts) `addSignedInt: numeral * numeral -> numeral`,
   `subSignedInt: numeral * numeral -> numeral`,
   `mulSignedInt: numeral * numeral -> numeral`,
   `divSignedInt: numeral * numeral -> numeral`,
   and `modSignedInt: numeral * numeral -> numeral`,
   for the arithmetic operators over signed integers; they must agree with the arithmetic operators defined for SML `int`s.
3. (3pts) `ltSignedInt: numeral * numeral -> bool`
   and `eqSignedInt: numeral * numeral -> bool`,
   for the relational operators over signed integers; they must agree with the relational operators defined for SML `int`s.

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

Use the following representation for rational numbers:

```
type rat = int * 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`.

3.8 Extra credit (25pts)

Consider the matrix $\mathbf{T}_{m,n}$ of homework exercise 1.6.2.

1. (5pts) Determine the closed form for the entries of $\mathbf{T}_{m,n}$.
2. (10pts) Using the type `type realmatrix = real list list` to represent real matrices, write a function `det: realmatrix -> real` to compute the determinant of a matrix, *using cofactor expansion*.
3. Let $\mathbf{S}_{m,n,j}$ be an $n \times n$ submatrix of $\mathbf{T}_{m,n}$ consisting of n consecutive rows of $\mathbf{T}_{m,n}$ starting with the j-th row.
   (a) (5pts) Show that $\det \mathbf{S}_{m,n,j}$ is the same for all j.
   (b) (5pts) Determine the closed form for $\det \mathbf{S}_{m,n,j}$.

How to turn in

Submission instructions will be posted to the 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.