Homework 3 — Third set of ML core language programs — assigned Monday 24 February, due Wednesday 5 March

Total number of points available on this homework is 200. Full credit is equivalent to 100 points.

Many problems in this homework set are extensions of problems from the first two sets. Therefore you can reuse your code. Try, however, to make the code more elegant, and more in the ML style of programming; use the full wealth of purely functional ML programming constructs.

Reading assignment

Read Chapters 5 and 6 of *ML for the Working Programmer*.

Optional reading: Read Chapter 10 of *The Little MLer*.

3.1 Minimum Spanning Trees (40pts)

You are given an undirected graph described by a value of type `(int * (int * int) list) list`.

Each element of this list is of the form (vi, [(vj,cij),(vk,cik),...,(vz,ciz)]) where vj, vk,..., vz are the vertices (nodes) that vertex vi is connected to, and cij, cik,..., ciz are correspondingly the costs of the edges between them.

Write a function `minSpanTree : (int * (int * int) list) list -> tree` which takes the graph (as described above) as argument and returns a minimum spanning tree of this graph using Prim's algorithm.

The tree is represented using this type:

```
datatype tree = Node of int * tree list
```

You may assume that the graph is connected and the input association list is complete for each vertex.

This link has a good explanation of Prim's algorithm:

`http://www.cs.ust.hk/faculty/tklove/COMP271/L11.pdf`

This link has a nice Java Applet where you can watch the algorithm work step-by-step:

`http://www-b2.is.tokushima-u.ac.jp/~ikeda/suuri/dijkstra/Prim.shtml`

3.2 Arithmetic Expressions (40pts)

3.2.1 Expression evaluator (20pts)

This is a continuation of exercises 2.2 and 2.3.

We can use the following data type declaration to introduce a language of simple arithmetic expressions, with variable names:

```
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 returns the arithmetic value of an expression (which may have free variables) in an environment (a mapping from variables to `int` values).

Then you can define:

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

so that `eval` evaluates closed expressions as before.

3.2.2 Expression parser (20pts)

Write a function `parse`, with type `string -> expr`, where `expr` is as above. The function should return an expression corresponding to the text of the string, in accordance with the standard precedence of arithmetic. The input string may consist of identifiers (which are as in ML), the keywords `let`, `in`, and `end`, the = sign, digits, +, -, *, and / signs, parentheses, and white space. For instance, `parse "5 + 3 * 4"` should evaluate to `Add (Num 5, Mul (Num 3, Num 4))`. In case the input string is not well-formed (for example, the string `"5 + 3 ) * 4"` is not), function `parse` should raise an exception `Failedbecause` (see p. 135 of *ML for the Working Programmer*).

Here is an example that uses let-expressions:
`val s = "5 + 3 * let x = 3 + (4 - 7) in let z'=2*(x-5) in x*x/z' end + 2 end / 2"`.
Given this declaration of the string `s`, the ML expression `eval (parse s)` should evaluate to `8`.

3.3 Using lists for arithmetic (50pts)

This is a continuation of exercises 1.5 and 2.4.

We introduce the following type `longint` to represent arbitrarily large integers:

```
datatype sign = Pos | Neg | Zero
datatype longint = Long of {radix: int, sign: sign; digits: int list}
exception LongDivByZero
```

Unlike in the previous exercises, long integers can be negative.

Write the functions:

- (10pts) `changeRadix: int -> longint -> longint`, such that `changeRadix r a` converts the long integer `a` to a new radix `r`.
- (10pts) `addLongInts: longint * longint -> longint` computes the sum of two integers; if the two operands have the same radix, that radix is used for the result as well, but if they do not have the same radix, then radix 10 is used for the result.
- (10pts) `subLongInts: longint * longint -> longint` computes the difference of two integers.
- (10pts) `mulLongInts: longint * longint -> longint` computes the product of two integers.
- (10pts) `divLongInts: longint * longint -> {quot: longint, rem: longint}` computes the quotient and the remainder and packages them as a record; exception `LongDivByZero` should be raised when appropriate.

3.4 Tic-tac-toe (40pts)

This is a continuation of exercise 2.5.

The game board for the game of tic-tac-toe is shown in Figure 1. The two players, X and O, alternate in claiming the spaces on the board; X goes first. The game ends when all the spaces have been claimed or as soon as a player has claimed three spaces in a line, in which case that player has won the game.

1	**2**	**3**
4	**5**	**6**
7	**8**	**9**

Figure 1: The tic-tac-toe game board.

We consider the question of play strategies for player X. We would like to construct a recipe that player X should follow so as not to lose (if possible), and preferably to win (if possible).

A strategy must contain a particular first move (since X goes first), and a particular response to each subsequent possible move of the opponent O.

A strategy can be understood as a mapping from inputs to outputs. Thus, X's response to no input at all is its first move, for instance, 5. Further on, X's response to O's input, for instance, 1, is X's second move, for instance, 4. Then, X's response to O's input, for instance 2 following 1, is X's third move, for instance 3. So we could say that X's strategy includes the rules to map the empty input [] to 5, the input [1] to 4, and the input [1,2] to 3.

We can represent a complete strategy for X, which is a set of such inputs-output pairs covering the whole game, using the ML type `(int list * int) list`.

Clearly, not all values of this type are valid strategies. How many valid strategies are there?

Among the valid strategies, some are *favorable* to X; namely, they do not specify any games in which X loses. How many favorable strategies are there?

Construct some favorable strategies.

3.5 Transmission codes (30pts)

This is a continuation of exercise 1.6.

Use the algorithm you developed in exercise 1.6 to construct a safe subset that is as large as possible, for the following values of the parameters: $K = 4$, $N = 18$, $D = 4$. Use the alphabet $\Sigma = \{0, 1, 2, 3\}$.

Points will be awarded according to the size of the safe subset computed, as follows. Among all submitted solution, all sets will be checked, and all sizes will be sorted in descending order. The largest set will win 30 points, the second largest 25 points, the third largest 20 points; the remaining sets will win 19,18,...,2,1,0 points, respectively.

How to turn in

Turn in your code by running

~roshan/handin your-file

on a regular UNM CS machine.

You should use whatever filename is appropriate in place of your-file. You can put multiple files on the command line, or even directories. Directories will have their entire contents handed in, so please be sure to clean out any cruft.