Homework 4 — ML core language — assigned Thursday 23 March — due Sunday 2 April

4.1 Polymorphic arithmetic (50pts)

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:

4.1.1 (1pt)

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

4.1.2 (1pt)

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

4.1.3 (1pt)

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

4.1.4 (1pt)

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

4.1.5 (1pt)

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

4.1.6 (1pt)

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

4.1.7 (3pts)

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

4.1.8 (3pts)

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

4.1.9 (5pts)

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

4.1.10 (10pts)

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

4.1.11 (5pts)

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

Define a type `rat` to represent rational numbers, along with its operations, as in homework exercise 3.7:

4.1.12 (2pts)

Declare a value `ratorderedfield: rat orderedfield`, to collect the various operations on rationals.

Test your polymorphic functions for linear algebra using these two examples:

4.1.13 (4pts)

The built-in reals, `realorderedfield`.

4.1.14 (4pts)

The rationals, `ratorderedfield`.

Redo homework exercise 1.6.3.5, this time using rationals:

4.1.15 (5pts)

Write a function `mixcomp: int * int -> rat lmatrix`.

4.2 An arithmetic expression translator: writing recursive functions over algebraic datatypes (50pts) [C; K.1.1; K.2.3; K.2.4; K.2.7]

In this exercise we use the same SML data type as in homework exercise 3.5. The goal is to translate an expression, which may have free variables, into a C program that evaluates the expression. If the expression has free variables, the C program queries them from `stdin`. For instance, the expression `Add (Var "x", Var "a")` should be translated into the C program

```
#include <stdio.h>
int main (int argc, char ** argv)
{
  int a;
  int x;
  int _t_1_;
  int _t_2_;
  int _t_3_;
  int _result_;

  printf ("a?\n");
  scanf ("%d", &a);
  printf ("x?\n");
  scanf ("%d", &x);

  _t_1_ = x;
  _t_2_ = a;
  _t_3_ = _t_1_ + _t_2_;
  _result_ = _t_3_;

  printf ("result=%d\n", _result_);
}
```

The C program queries the user for the values of the free variables of the expression in alphabetical order.

Any subexpression that can be evaluated independent of the values of free variables must be evaluated by the translator, not by the C program. To make this more precise: if and only if the arguments to an operator do not depend on any free variables, we consider that the result of the operator does not depend on any free variables. (This means that we do not use algebraic knowledge to simplify, e.g., $x - x$ to 0.)

The C program uses temporary variables `t`*n*, one for each node of the expression tree that could not be immediately evaluated by the translator; the right-hand side of each assignment statement is either one of the free variables or a C binary arithmetic operator (`+`,`-`,`*`,`/`) where each operand is either a temporary or a constant. Finally, the temporary corresponding to the root of the expression tree is assigned to `result`, and that is printed.

Write a function `translate: expr -> string` that converts an expression into a string containing the appropriate C program.

Additional examples

- `Let {body=Add (Var "x",Var "x"),value=Add (Num 5,Var "x"),var="x"}`

```
#include <stdio.h>
int main (int argc, char ** argv)
{
  int x;
  int _t_1_;
  int _t_2_;
  int _t_3_;
  int _result_;

  printf ("x?\n");
  scanf ("%d", &x);

  _t_1_ = x;
  _t_2_ = 5 + _t_1_;
  _t_3_ = _t_2_ + _t_2_;
  _result_ = _t_3_;

  printf ("result=%d\n", _result_);
}
```

- `Let {body=Add (Num 3,Num 6),value=Add (Num 5,Var "x"),var="x"}`

```
#include <stdio.h>
int main (int argc, char ** argv)
{
  int x;
  int _result_;

  printf ("x?\n");
  scanf ("%d", &x);

  _result_ = 9;

  printf ("result=%d\n", _result_);
}
```

- `Add (Let {body=Add (Var "x",Var "x"),value=Add (Num 5,Var "x"),var="x"},`
  `Add (Var "x",Var "a"))`

```
#include <stdio.h>
int main (int argc, char ** argv)
{
  int a;
```

```
  int x;
  int _t_1_;
  int _t_2_;
  int _t_3_;
  int _t_4_;
  int _t_5_;
  int _t_6_;
  int _t_7_;
  int _result_;

  printf ("a?\n");
  scanf ("%d", &a);
  printf ("x?\n");
  scanf ("%d", &x);

  _t_1_ = x;
  _t_2_ = 5 + _t_1_;
  _t_3_ = _t_2_ + _t_2_;
  _t_4_ = x;
  _t_5_ = a;
  _t_6_ = _t_4_ + _t_5_;
  _t_7_ = _t_3_ + _t_6_;
  _result_ = _t_7_;

  printf ("result=%d\n", _result_);
}
```

- Mul

```
     (Add
        (Let {body=Add (Var "x",Var "x"),value=Add (Num 5,Var "x"),var="x"},
         Add (Var "x",Var "a")),
      Sub
        (Let {body=Add (Var "x",Var "x"),value=Add (Num 5,Var "x"),var="x"},
         Div (Add (Var "x",Var "a"),Add (Var "x",Var "a"))))
```

```
#include <stdio.h>
int main (int argc, char ** argv)
{
  int a;
  int x;
  int _t_1_;
  int _t_2_;
  int _t_3_;
  int _t_4_;
  int _t_5_;
  int _t_6_;
```

```
    int _t_7_;
    int _t_8_;
    int _t_9_;
    int _t_10_;
    int _t_11_;
    int _t_12_;
    int _t_13_;
    int _t_14_;
    int _t_15_;
    int _t_16_;
    int _t_17_;
    int _t_18_;
    int _t_19_;
    int _result_;

    printf ("a?\n");
    scanf ("%d", &a);
    printf ("x?\n");
    scanf ("%d", &x);

    _t_1_ = x;
    _t_2_ = 5 + _t_1_;
    _t_3_ = _t_2_ + _t_2_;
    _t_4_ = x;
    _t_5_ = a;
    _t_6_ = _t_4_ + _t_5_;
    _t_7_ = _t_3_ + _t_6_;
    _t_8_ = x;
    _t_9_ = 5 + _t_8_;
    _t_10_ = _t_9_ + _t_9_;
    _t_11_ = x;
    _t_12_ = a;
    _t_13_ = _t_11_ + _t_12_;
    _t_14_ = x;
    _t_15_ = a;
    _t_16_ = _t_14_ + _t_15_;
    _t_17_ = _t_13_ / _t_16_;
    _t_18_ = _t_10_ - _t_17_;
    _t_19_ = _t_7_ * _t_18_;
    _result_ = _t_19_;

    printf ("result=%d\n", _result_);
  }
```

- `Let`
  `{body=Let {body=Add (Var "x",Var "a"),value=Num 3,var="a"},value=Num 6,`

```
     var="x"}

#include <stdio.h>
int main (int argc, char ** argv)
{
  int _result_;


  _result_ = 9;

  printf ("result=%d\n", _result_);
}
```

- Let

```
    {body=Div (Var "x",Var "x"),
     value=Let
             {body=Let {body=Add (Var "x",Var "a"),value=Num 3,var="a"},
              value=Num 6,var="x"},var="x"}

#include <stdio.h>
int main (int argc, char ** argv)
{
  int _result_;


  _result_ = 1;

  printf ("result=%d\n", _result_);
}
```

- Sub (Var "x",Var "x")

```
#include <stdio.h>
int main (int argc, char ** argv)
{
  int x;
  int _t_1_;
  int _t_2_;
  int _t_3_;
  int _result_;

  printf ("x?\n");
  scanf ("%d", &x);

  _t_1_ = x;
  _t_2_ = x;
```

```
  _t_3_ = _t_1_ - _t_2_;
  _result_ = _t_3_;

  printf ("result=%d\n", _result_);
}
```

- Let

```
    {body=Let {body=Div (Var "x",Var "y"),value=Add (Var "x",Num 9),var="y"},
     value=Let
             {body=Div (Var "x",Var "x"),
              value=Let
                      {body=Let
                              {body=Add (Var "x",Var "a"),value=Num 3,var="a"},
                       value=Num 6,var="x"},var="x"},var="x"}
```

```
#include <stdio.h>
int main (int argc, char ** argv)
{
  int _result_;


  _result_ = 0;

  printf ("result=%d\n", _result_);
}
```

- Let

```
    {body=Let {body=Div (Var "z",Var "y"),value=Add (Var "x",Num 9),var="y"},
     value=Let
             {body=Div (Var "x",Var "x"),
              value=Let
                      {body=Let
                              {body=Add (Var "x",Var "a"),value=Num 3,var="a"},
                       value=Num 6,var="x"},var="x"},var="x"}
```

```
#include <stdio.h>
int main (int argc, char ** argv)
{
  int z;
  int _t_1_;
  int _t_2_;
  int _result_;

  printf ("z?\n");
  scanf ("%d", &z);
```

```
  _t_1_ = z;
  _t_2_ = _t_1_ / 10;
  _result_ = _t_2_;

  printf ("result=%d\n", _result_);
}
```

4.3 Graduate credit: An arithmetic expression translator (continuation): writing recursive functions over algebraic datatypes (20pts) [C; K.1.1; K.2.3; K.2.4; K.2.7]

This problem is required for graduate credit; other course participants may solve this problem for extra credit.

Repeat the preceding problem, but use all applicable algebraic knowledge to simplify expressions. For instance, $x - x$ can be simplified to 0 even if we do not know the value of x.

How to turn in

Submission instructions have been 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.