CSci 4651 Fall 2003
Problem Set 8: Modules.
Due Monday, November 24st

**Problem 1 (ML abstype).** Consider the following specification of fractions: a fraction is a pair of two integers (the numerator and denominator) which has the following operations:

1. `denom` – returns the denominator of the fraction,
2. `numer` – returns the numerator of the fraction,
3. `add` – adds two fractions,
4. `mult` – multiplies two fractions.

Fractions don't have to be simplified, so $\frac{2}{4}$ is a valid fraction.

**Question 1.** Write an implementation of fractions as an abstract type `frac` which stores the denominator and the numerator as a pair of integers: `int * int`. Note that you need to provide a constructor for a fraction.

Test your implementation using the following code (available in my pub directory on epoxy in the file `ps8_test`):

```
val half = frac(1,2);
val third = frac(1,3);
val fivesixth = add(half, third);
numer(fivesixth);
denom(fivesixth);
val onesixth = mult(third,half);
numer(onesixth);
denom(onesixth);
```

**Question 2.** Write another implementation of fractions as an abstract type `frac1` which stores the denominator and the numerator as the following record: `{num:int,den:int}`. Recall that a value of this type is pattern-matched as `{num=x,den=y}`. Test this implementation on the same test code (replacing `frac` by `frac1`).

**Question 3.** Briefly explain why the two implementations have the same behavior.

**Problem 2 (modules).** This problem uses notations similar to Modula-2. Study examples on p. 254 for this problem. While the syntax in this example is very close to that of Modula-2, it may not be the exact Modula-2 syntax. Please ignore the differences. If the syntax is unclear, please ask.

Consider the following two implementation modules:

Implementation A (using an array of two elements):

```
implementation module Fractions;
type Fraction = Array[1..2] OF INTEGER;
Procedure Add (x, y : Fraction) : Fraction;
   VAR temp: Fraction;
   BEGIN
       temp[1] := x[1] * y[2] + x[2] * y[1];
       temp[2] := x[2] * y[2];
       RETURN temp;
   END Add;
Procedure Mult (x, y : Fraction) : Fraction;
   VAR temp: Fraction;
   BEGIN
       temp[1] := x[1] * y[1];
       temp[2] := x[2] * y[2];
       RETURN temp;
   END Mult;
Procedure Newfrac (x, y : INTEGER) : Fraction;
   VAR temp: Fraction;
       temp[1] := x;
       temp[2] := y;
       RETURN temp;
   END Newfrac;
END Fractions.
```

Implementation B (using records):

```
implementation module Fractions;
type Fraction = RECORD
                    num: INTEGER;
                    den: INTEGER;
                END;
Procedure Add (x, y : Fraction) : Fraction;
   VAR temp: Fraction;
   BEGIN
       temp.num = x.num * y.den + x.den * y.num;
       temp.den = x.den * y.den;
       RETURN temp;
   END Add;
Procedure Mult (x, y : Fraction) : Fraction;
   VAR temp: Fraction;
   BEGIN
       temp.num = x.num * y.num;
       temp.den = x.den * y.den;
       RETURN temp;
   END Mult;
Procedure Newfrac (x, y : INTEGER) : Fraction;
```

```
   VAR temp: Fraction;
       temp.num := x;
       temp.den := y;
       RETURN temp;
   END Newfrac;
END Fractions.
```

Consider the following three interfaces:
Interface 1:

```
definition module Fractions;
    type Fraction;
    procedure Add (x, y : Fraction) : Fraction;
    procedure Mult (x, y : Fraction) : Fraction;
    procedure Newfrac(x, y: integer) : Fraction;
end Fractions;
```

Interface 2:

```
definition module Fractions;
    type Fraction;
    procedure Add (x, y : Fraction) : Fraction;
end Fractions;
```

Interface 3:

```
definition module Fractions;
    procedure Add (x, y : Fraction) : Fraction;
    procedure Mult (x, y : Fraction) : Fraction;
    procedure Newfrac(x, y: integer) : Fraction;
end Fractions;
```

Consider the following three main modules:
Main I:

```
module main
IMPORT Fractions; (* importing the entire module*)
VAR x, y, z, w : Fraction;
BEGIN
   x = Newfrac(1,2);
   y = Newfrac(1,3);
   z = Add(x,y);
   w = Mult(x,y);
END main;
```

Main II:

```
module main
FROM Fractions IMPORT Fraction, Add, Newfrac;
VAR x, y, z, w : Fraction;
BEGIN
   x = Newfrac(1,2);
   y = Newfrac(1,3);
   z = Add(x,y);
END main;
```

Main III:

```
module main
FROM Fractions IMPORT Add, Newfrac;
VAR x, y, z, w : Fraction;
BEGIN
   x = Newfrac(1,2);
   y = Newfrac(1,3);
   z = Add(x,y);
END main;
```

Of all possible combinations of the two implementations, three interfaces, and three main modules, which ones will and which ones will not work? Please explain your answer briefly. You may represent your answer as a table.

**Problem 3 (ML signatures and structures).** *Note:* all code shown for this problem is available in my pub directory on epoxy in the file struct.sml.

Consider the following structure

```
signature SET =
sig
type elem
type set
val empty: set
val isMember: elem * set -> bool
val allMembers : set -> elem list
val add: elem * set -> set
end;
```

The function `allMembers` returns a list of all elements of the set.

**Question 1.** Write two implementations of a set of integers:

**Implementation 1.** An implementation `struct set1` that stores elements as a list. A new element is appended at the beginning of the list.

**Implementation 2.** An implementation `struct set2` that uses a binary search tree to store elements. The first element gets stored at the root of the tree. Each new element is compared to the root. If it is less than the root, it is inserted into the left subtree of the root. If it is greater than the the root, it is inserted into the right subtree of the root. If it equals to the root, then we don't need to insert it at all: it's already a member of the set. The procedure is repeated recursively for the subtree in which the element is being inserted.

Below is the datatype of the tree (study it carefully: it is different from the datatype used in an earlier problem set):

```
datatype tree = EMPTY | LEAF of int | NODE of int * tree * tree;
```

It might be helpful to write all the necessary tree functions before writing the set implementation. The function `find` below will be helpful for implementing `isMember`:

```
fun find (n, EMPTY) = false |
    find (n, LEAF(m)) = if (n = m) then true else false |
    find (n, NODE(m, t1, t2)) = if (n = m) then true else
if (n < m) then find(n, t1) else find(n, t2);
```

The built-in operator for appending two lists might be helpful. It works as follows:

```
[4,5]@[8,7];
```

produces the list [4,5,8,7].

Test both implementations using the following code (replacing `set` by `set1` or `set2`, depending on the implementation that you are testing):

```
val s = set.empty;
val s1 = set.add(2, set.add(5, set.add(1, set.add(7,s))));
set.isMember(2,s1);
set.isMember(3,s1);
set.isMember(5,s1);
set.isMember(7,s1);
set.isMember(1,s1);
set.isMember(9,s1);
set.isMember(6,s1);
set.allMembers(s1);
```

**Question 2.** Compare the two implementations from the point of view of efficiency, convenience, and ease of adding new features. In particular, which implementation will be more convenient to extend with a function `size` which returns the number of elements in the set?

Is one of the implementation always better than the other, or does it depend on the intended use? Please explain your answer.