Advanced Models and Programs
Carsten Schürmann
carsten@itu.dk

Homework 3

Please hand in a printout of your code by
Wednesday, 21 February 2007
before class meets.
Extensions can *only* be granted before the deadline
(if necessary, contact the teaching staff by email in English, please.)

Guidelines

While we acknowledge that beauty is in the eye of the beholder, you should nonetheless strive for elegance in your code. Not every program which runs deserves full credit. Make sure to state invariants in comments which are sometimes implicit in the informal presentation of an exercise. If auxiliary functions are required, describe concisely what they implement. Do not reinvent wheels, and try to make your functions small and easy to understand. Use tasteful layout and avoid long winded and contorted code. On the website, you will find a file called "blueprint.sml" that contains all code (including queries), that you need to complete this assignment.

Problem [100]

In this homework problem, we will develop an interpreter for a dialect of SQL in SML/NJ which we will call miniSQL. Rest assured, we don't have any intentions of writing an interpreter for a complete query language, but it might be suprising of how powerful that little language actually is.

Let's begin with what data actually is. In relational databases theory, tables are nothing else then relations. In general, a table can have many columns, and every column usually carries information how the data is supposed to be interpreted. In fact, this is extremely important, as data is usally stored on a harddrive or in memory as a sequence of bits. In miniSQL, we'll distinguish between four cases of how to represent data: There are three base cases, which define integers, string, and reals respectively, and then their is a pairing case, which allows us to consider several columns at the same time. A list of values of type `Data` forms a `Table`

```
datatype Data =
  Int of int
  | String of string
  | Real of real
  | Pair of Data * Data
type Table = Data list
```

For example a row in a table that contains chemical elements, might contain the name of the element, the number in the periodic table, and the melting point in degree Celsius.

```
val ChemicalElements : Table
       = [Pair (Pair (String "H", Int 1), Real ~259.14),
          Pair (Pair (String "He", Int 2), Real ~272.00),
          Pair (Pair (String "Li", Int 3), Real 180.54),
          Pair (Pair (String "Be", Int 4), Real 1278.00),
          Pair (Pair (String "O", Int 8), Real ~218.36)]
```

As a second example, consider a set of rows, where the first element names a person, and the second and third father and mother, respectively.

```
val Parents : Table
  = [Pair (String "Mary", Pair (String "George", String "Alice")),
     Pair (String "Bob", Pair (String "George", String "Alice")),
     Pair (String "George", Pair (String "Robert", String "Hillary")),
     Pair (String "Alice", Pair (String "Max", String "Lena")),
     Pair (String "Peter", Pair (String "Max", String "Lena")),
     Pair (String "Lena", Pair (String "Rudolph", String "Magret"))]
```

In general, we introduce a notational shortcut an write `Pair (Pair (String, Int), Real)` for the "kind" of the table with the chemical elements, and `Pair (String, Pair (String, String))` for the kind of table with the child parent relationship. Note, that this is just an informal abbreviation, you cannot type this into ML.

MiniSQL

The SQL language is famous for its brief and descriptive syntax: `select` some fields `from` some tables `where` some kind of condition holds. The `from` clause permits the joining of tables. When we join two tables we append each reach from the first table to each row of the second table and throw them together into a new table. Assuming the given tables have kind A and B, respectively, the joined table will be of kind `Pair` (A, B). The select clause allows us to match records in a table and drop, insert, or exchange columns in the table and yields another table. The `where` clause allows us to specifiy the properties that the rows of the resulting table need to specify. Operationally speaking, we can thing of the `where` clause as a filter operation, that takes a table and removes all rows from it, that do not match the condition.

We will now describe the expression language of MiniSQL and how each of the cases is supposed to work. The datatype given here, describes all expressions of the language. For now, ignore the `Load` and `Save` expression constructor, we will explain them later.

```
datatype Exp =
    From of Table * Exp
  | Dup of Exp
  | Join of Exp
  | Where of (Data -> bool) * Exp
  | Iterate of (Data * Table -> Table) * Exp
```

```
  | Select  of (Data -> Data) * Exp
  | Load of string * layout * Exp
  | Save of string * layoutInverse * Exp
  | End
```

Intuitively, `From` introduces the table we would like to operate on. As we have seen, we might want to join two tables to form a new table. The way we do this in miniSQL is inspired by the reverse polish notation for numbers. If we want to evaluate the expression $5*(3+4)$, for example, we write 5 3 4 + *. Note, that this expression does not need any parenthesis. When we evaluate it, we do this left to right. We need to imagine that we also have a stack, where we stack up numbers. Here is how we evaluate the expression.

5 3 4 + *	put 5 on the stack
3 4 + *	put 3 on the stack
4 + *	put 4 on the stack
+ *	take the top two elements from the stack (4, 3), add, and put the result 7 back onto the stack
*	take the top two elements from the stack (7,5), multiply and put the result 35 back onto the stack
35	since nothing more needs to be done.

We use the same trick in miniSQL, except that we need to put tables on the stack, and not just numbers. Now everything will fall into place.

```
datatype Stack =
  Null
  | Cons of Stack * Table
```

We consider each one of the constructors for miniSQL expression and explain what they do. Note, that the last argument to all constructors but `End` is an expression. It always stands for the rest of the miniSQL program still to be executed.

From Put `Table` onto the stack. Continue with evaluating `Exp`.

Dup Make a copy of the top `Table` of the stack and push it onto the stack, before you continue evaluating `Exp`.

Join Compute the join of the two top `Table`s of the stack. Continue with evaluating `Exp`.

Where The first argument to `Where` is a predicate. Take the top `Table` from the stack, filter out all those rows that satisfy the predicate, and put the resulting `Table` back onto the stack. Continue with evaluating `Exp`.

Iterate The iterator corresponds to a fold on the top `Table` of the stack. The first argument is a function that expects as an argument the current row of data and the result of having iterated over the rest of the `Table`. The result of the iterator is another `Table` that replaces the old one on top of the stack. Continue with evaluating `Exp`.

Select The selector corresponds to a map, and will be applied to all rows of the `Table`. The result is another `Table` taht replaces the old one on the top of the stack. Continue with evaluating `Exp`.

End Return the top `Table` of the stack and terminate.

Problem 1:

Implement the filter function that you will need for the `select` case in Problem 3.

```
filter : Table -> (Data -> Bool) -> Table
```

Problem 2:

Implement the join function that you will need for the `join` case in Problem 3.

```
join : Table -> Table -> Table
```

Problem 3:

Implement the run funcion that will execute a query as outlined above.

```
run : Stack -> Expression -> Table
```

Example 1:

Now we look at some examples. On which you can test your code. Let

```
val T0 : Table
       = map (fn x => Int x) [1,2,3,4,5,6,7,8,9]
```

a table of 9 numbers. We consider a few queries in turn.

```
val E1 : Table
       = run Null (From (T0,
                  Iterate (fn (Int x, []) => [Int x]
                            | (Int x, [Int r]) => ([Int (x+r)]),
                  End)))
```

returns `val E1 = [Int 45] : Table`.

```
(* Example 2: Compute the sum of all even numbers in T0 *)
val E2 = run Null (From (T0,
                  Where (fn (Int x) => x mod 2 = 0,
                  Iterate (fn (Int x, []) => [Int x]
                            | (Int x, [Int r]) => [Int (x + r)],
                  End))))
```

returns val E2 = [Int 20] : Table.

```
(* Example 3: Compute the list of all squares that are odd *)
val E3 = run Null (From (T0,
                  Iterate (fn (Int x, r) => if x * x mod 2 = 0 then r else Int x :: r,
                  End)))
```

returns `val E3 = [Int 1,Int 3,Int 5,Int 7,Int 9] : Table`.

Problem 4:

Write a query that computes the multiplication table for the the table `T0` defined above.

Databases are usually persistently stored on hardrives etc. To make miniSQL a little more like the usual database query language, we also implement the functionality saving and loading datafiles. The type `Data` is a semantically enriched way of representing the bytes that are stored in a file. Therefore, we introduced the concept of a `layout`,

```
type layout = string -> Data
type layoutInverse = Data -> string
```

that interprets strings as `Data`, and vice versa, a `layoutInverse` interprets Data of string. Both are interpreted as functions, of course. Now we can add two more constructors to miniSML:

```
datatype Exp =
    ...
  | Load of string * layout * Exp
  | Save of string * layoutInverse * Exp
```

The implementation of a load and save function are given. When you implement the interpreter for miniSQL, please just call them.

Example 2:

Consider the table `Parents` of a set of fictitious persons, and their parents.

```
val Parents : Table
  = [Pair (String "Mary", Pair (String "George", String "Alice")),
     Pair (String "Bob", Pair (String "George", String "Alice")),
     Pair (String "George", Pair (String "Robert", String "Hillary")),
     Pair (String "Alice", Pair (String "Max", String "Lena")),
     Pair (String "Peter", Pair (String "Max", String "Lena")),
     Pair (String "Lena", Pair (String "Rudolph", String "Magret"))]
```

The following query computes a table of kind `Pair (String, Pair (Pair (String, String), Pair (String, String)))` which is a table relating grandchildren to their four grandparents.

```
(* Example 6: Compute the grandparent relationship from the parent relationship
              Table of records of the form (c, ((fm, ff), (mf, mm))  *)
val E6 = run Null (From (Parents,
```

```
Dup (
Dup (
Join (
Join (
Where (fn (Pair (Pair (String c1, Pair (String f1, String m1)),
                 Pair (Pair (String c2, Pair (String f2, String m2)),
                       Pair (String c3, Pair (String f3, String m3)))))
       => f1 = c2 andalso m1 = c3,
Select (fn (Pair (Pair (c1, Pair (String f1, String m1)),
                 Pair (Pair (String c2, Pair (f2, m2)),
                       Pair (String c3, Pair (f3, m3)))))
       => Pair (c1, Pair (Pair (f2, m2), Pair (f3, m3))),
Save ("grandparents",
      fn (Pair (String c1, Pair (Pair (String f2, String m2),
                                 Pair (String f3, String m3)))) =>
      c1 ^ " grandparents are " ^ f2 ^ ", " ^ m2 ^ ", " ^ f3 ^ ", "
         ^ m3 ^ ".\n",
End)))))))))
```

Note the use of a layout here.

Problem 5:

Make sure that your interpreter runs on this query. You can expect the result to be a file that is called ‘‘`grandparents`’’. What does it contain?

Problem 6:

On the website you can find three files, named `dkk-dollar`, `dollar-euro`, and `euro-dkk`, that contain daily midday exchange courses for the respective currencies starting from January 1, 2006 all the way until last week. In this problem, we want to use miniSQL, in order to determine if we could make money by exchanging DKK's into dollars, then into Euro's and then back into DKK's, using the mid-day rates of each day.

The layout of the date in the files is given by the two functions:

```
val LayoutExchangeRate : layout
                       = fn s => Pair (String (String.substring (s, 0, 10)),
                                       Real (valOf (Real.fromString
                                                    (String.substring (s, 20, 10)))))
val LayoutExchangeRateInverse : layoutInverse
      = fn (Pair (String s, Real r))
      => s ^ "          "
           ^ String.substring (Real.toString (r) ^ "0000000000", 0, 10)
           ^ "\n";
```

Here is a sample query:

```
val E7 = run Null (Load ("dkk-dollar", LayoutExchangeRate,
                  Load ("dollar-euro", LayoutExchangeRate,
```

```
Join (
Where (fn (Pair (Pair (String d1, _), Pair (String d2, _)))
       => d1 = d2,
Select (fn (Pair (Pair (d1, Real r1), Pair (_, Real r2)))
        => Pair (d1, Real (r1 * r2)),
Save ("my-result", LayoutExchangeRateInverse,
End)))))))
```

will return a file `my-result` which is formatted just the same as the given files, but it contains the exchange between dkk and euros.

Please write a query, that computes the daily rate from DKK to DKK that one would get by first exchanging the money into dollars and then into Euros. Save the result in a file called “dkk-dkk”.