Computer Science 320 (Fall, 2008)
Concepts of Programming Languages

Assignment 2: Higher-order List Functions

Out: Tuesday, September 15, 2008
Due: Friday, September 25, 2008

This assignment is divided into two parts; each part demonstrates an application of functional higher-order list functions. All your solutions should be defined within two modules: `RelDB` and `Superstring`. You should submit the files `Superstring.hs` and `RelDB.hs` (or `*.lhs`). **File names are case sensitive.**

Note: All definitions at top level must have explicit type annotations. Unless otherwise specified, solutions may utilize functions from the standard prelude and any material in the course notes and textbook. You may **not** use any other materials without explicit citation.

1 Relational Databases

Many modern implementations of relational databases (such as the family of SQL servers) provide functionality on tables that corresponds to the higher-order list functions commonly used in functional programming.

Figure 1: Typical relational database table structure.

	column name	column name	column name
row 1	value	value	value
row 2	value	value	value
⋮	⋮	⋮	⋮

For this part of the assignment, you should start with the following module definition. Table values will be modelled using an algebraic data type, and a database table (as illustrated in Figure 1) will be defined as a list of column names and a list of rows in which each row is a list of values. The lengths of all rows should match the number of columns.

```
module RelDB where

data Value = I Int | S String | Null
  deriving Show --so that you can display values

type Column = String
type Row = [Value]
type Table = ([Column], [Row])
```

Problem 1. (10 pts)

First, you will define several operators on values. Add the following type definitions to your module.

```
type Op = Value -> Value -> Value
type LogicOp = Value -> Value -> Bool
type AggregateOp = [Value] -> Value
```

(a) Define a function `plus :: Op` that computes the sum of two database values. If at least one argument is not an integer, `plus` should return `Null`.

(b) Define a function `ccat :: Op` that concatenates two database values. If at least one argument is not a string, `ccat` should return `Null`.

(c) Define a function `eqls :: LogicOp` that returns `True` if its two arguments are equivalent (even if both are `Null`) and `False` otherwise.

(d) Define a function `agg :: Op -> Value -> AggregateOp` that takes an operator `op` and a default value `base`, and returns an aggregate operator that performs the operation specified by `op` across an entire list of values. If the list of values supplied to the aggregate operator is empty, it should return the default value specified by `base`. Below are two examples of how this function should behave.

```
(agg plus (I 0))  [I 1, I 2, I 3, I 4] = I 10
(agg ccat (S "")) [S "a", S "b", S "c"] = S "abc"
```

Problem 2. (30 pts)

You will now implement several table operations commonly supported by implementations of relational databases.

(a) Define a function `v :: Column -> [(Column, Value)] -> Value` that takes a column name along with a list of values labelled by column, and returns the appropriate value (or `Null` if there is no such column in the list). You may assume that the column names in the list will be unique.

(b) Define a function `select :: [Column] -> Table -> Table` that takes a column list and a table, and returns a new table containing only the columns in the column list. It is possible to implement this function in one line using `zip` and list comprehensions.

(c) Define a function `aggregate :: AggregateOp -> Column -> Table -> Value` that takes an aggregate operator, a column, and a table. It should return the result of applying the aggregate operator over all the values in the specified column of the table.

(d) Define a function `join :: Table -> Table -> Table` that takes two tables and returns a table that has the columns from both tables (you may assume no column names are duplicated between the tables). The resulting table should have a row for every pair of rows in which one row is taken from the first table, and the other row is taken from the second table. That is, if the first table has n rows and the second table has m rows, the resulting table should have $m \cdot n$ rows.

(e) Add the following data type to your module.

```
data OpArg = C Column | V Value
```

Define a function `only :: OpArg -> LogicOp -> OpArg -> Table -> Table` that takes a logical operator, two arguments for that operator, and a table. It should return a table that contains only the rows for which the operator, when applied to the values (or column values) specified by the two arguments, returns `True`. The resulting table should contain the same columns as the input table.

Problem 3. (10 pts)

Add the following two tables to your module.

```
tbl1 = (
    ["Metro area",    "Population"], [
    [S "Tokyo",         I 32450000],
    [S "Seoul ",        I 20550000],
    [S "Mexico City",   I 20450000],
    [S "New York City", I 19750000],
    [S "Mumbai",        I 19200000],
    [S "Jakarta",       I 18900000],
    [S "Sao Paulo",     I 18850000],
    [S "Delhi",         I 18600000] ] )

tbl2 = (
    ["City",            "Country"], [
    [S "Tokyo",         S "Japan"],
    [S "Seoul ",        S "South Korea"],
    [S "Mexico City",   S "Mexico"],
    [S "New York City", S "United States"],
    [S "Mumbai",        S "India"],
    [S "Jakarta",       S "Indonesia"],
    [S "Sao Paulo",     S "Brazil"],
    [S "Delhi",         S "India"] ] )
```

(a) Using *only* the functions you were explicitly asked to define in **Problem #2**, define `tbl3 :: Table` so that it contains only the columns `"Country"` and `"Population"`, specifying the population of the largest city in the corresponding country.

(b) Using only the functions you were explicitly asked to define in **Problem #1** and **Problem #2**, define `population :: Value` to be the total population of the two most populous cities in India.

2 Shortest Superstring Problem

The shortest superstring problem is encountered in both DNA sequencing and data compression. The problem is defined in the following way: given a list of strings $S = \{s_1, \ldots, s_n\}$, what is the shortest possible string s^* such that every string $s_i \in S$ is a substring of s^*? You will use higher-order list functions to implement an algorithms for solving this problem.

Problem 4. (20 pts)

(a) Define a function `overlap :: (String, String) -> Int` that takes two strings s and s' and returns the number of elements at the end of s which overlap with the beginning of s'. For example:

```
  overlap ("fire", "red")  =  2
overlap ("water", "blue")  =  0
```

You may want to use the function `isPrefixOf`, found in the prelude. To use it, you will need to add `import List` immediately after your module declaration.

(b) Define a function `contains :: String -> String -> Bool` that returns `True` only if the second string is a substring of the first string. Otherwise, it returns `False`.

(c) Define an infix operator `o :: String -> String -> String` that concatenates two strings s and s' by merging the overlapping elements at the end of s and the beginning of s'. For example:

```
  "fire" ‘o‘ "red"  =  "fired"
"water" ‘o‘ "blue"  =  "waterblue"
```

(d) Using `foldr` and `(o)`, define a superstring algorithm `naive :: [String] -> String` that takes a list of strings, and returns a superstring containing every string in that list.

Problem 5. (30 pts)

Add the following higher-order function to your module. The function take an objective function `obj` and two arguments. It returns the argument that minimizes the objective function.

```
minimize :: (a -> Int) -> a -> a -> a
minimize obj x y = if obj x < obj y then x else y
```

You will now define the optimal superstring algorithm.

(a) Define a function `allPairs :: [String] -> [(String , String)]` that takes a list of strings, and returns a list of all pairs of strings that can be made out of this list's elements, *excluding* any pairs of equal strings.

(b) Using `filter`, define a function `update :: [String] -> (String, String) -> [String]` that takes a list of strings ℓ, and a pair of strings (s, s'). The function should combine the two strings s and s' (taking advantage of any overlapping) to obtain a new string s''. It should remove from the list of strings ℓ any strings contained in s'', and should then return a list that contains all of the remaining strings as well as one copy of s''. For example:

```
update ["fire", "red", "blue"] ("fire", "red")  =  ["fired","blue"]
```

(c) Define a recursive algorithm `superstring :: ([String] -> [(String, String)]) -> [String] -> String` that takes a function `next` for generating pairs of strings out of a list, and a list of strings ℓ. Given an empty list, `superstring` should return an empty list. Given a list with only one string, it should return that string. Otherwise, the algorithm should operate in the following manner at each invocation.

   (1) Take the list of strings and generate a list of pairs using the `next` function.
   (2) For *every* pair in the list from (1), generate a new list with *only* that pair combined (thus progressing the superstring computation one step closer to a solution).
   (3) Every list of strings generated in (2) is a smaller version of the superstring problem. The algorithm should call itself recursively to solve this smaller problem. Remember that any recursive call is also expecting `next` as an argument. This portion can be implemented using `map` or list comprehensions.
   (4) When the recursive calls all return a result, return the shortest (you can use `foldr`, `length`, and `minimize` for this; for the base case of `foldr`, you can use `naive`).

   This can be defined in a single line; plan carefully and use list functions when possible.

(d) Define a superstring algorithm `optimal :: [String] -> String` that takes a list of strings, tries all possible ways of combining the strings, and returns the best result.[1] Because it tries all possibilities, this algorithm always returns the shortest possible superstring, but runs in exponential time. Nevertheless, your implementation should be able to handle the following test cases:

```
test1  =  ["ctagcgacat", "aagatagtta", "gctactaaga", "gacatattgt", "tagttactag"]
test2  =  ["101001","010100010100", "100101", "001010", "11010", "100", "11010"]
test3  =  [x++y | x<-["aab","dcc","aaa"], y<-["dcc", "aab"]]
```

[1]Hint: You only need your solutions from parts (a) and (c) above.