CEng 242 Homework 3

Due: 6^{th} April 2001

An international car distribution company collects its orders in a special type called `CarOrder`. A `CarOrder` value consists of set of order items for cars where each order entry have:

Producer:	A `string` denoting the producer of the car like "Ford", "Opel", "Ferrari", "Honda" etc.
Model:	A `string` denoting the model of the car like "Sierra GT", "Astra", "TX200TC", "CLX" etc.
Year:	An `int` value greater than 1900 and less then or equal to 2001 which is the production year of the car.
Color:	Any `string` value denoting the color of the car like "Blue", "White", "Rain forest green", "Night shadow blue" etc.
Class:	A `CarClass` type value denoting the class the car belongs to. `CarClass` type declaration is given below.
Amount:	An `int` value greater then 0 representing how many cars of this type is ordered.
Unit price:	A `real` value greater then 0 representing the unit price of the car.

For each `CarOrder` typed value a tax strategy can be defined as follows: A default tax rate percentage which should be greater then or equal to 0 is kept. In addition to this, a list of tax rules is defined and applied incrementally for each order entry separately. A tax rule can be any of:

`TaxPrice(a,b,addtax)`
: If unit price UP is in the interval $\texttt{a} \leq UP < \texttt{b}$, add real value `addtax` to current tax percentage. $0 < \texttt{a} \leq \texttt{b}$ and $0 < \texttt{addtax}$ should hold for `a, b` and `addtax`.

`TaxClass(class,addtax)`
: If class of the order entry is `class` then add the real value `addtax` to current tax percentage. $0 < \texttt{addtax}$ should hold.

`TaxYear(a,b,dectax)`
: If production year of the cars in the order item in the closed interval `[a,b]` then the current tax will be reduced by the percentage value in real `dectax` value. New tax percentage will be $currenttax - currenttax * dectax/100$. $\texttt{a} \leq \texttt{b}$ and `dectax` < 100 should hold.

These rules are kept in the order of insertion and applied in this order starting with the default tax rate for each order item individually.
You are given some of the declarations and you should complete the abstract data type definition in the following:

```
datatype CarClass = Hatchback | Sedan | Convertible | StationWagon |
                        Jeep | Pickup | MiniBus ;
datatype TaxRule = TaxPrice of real*real*real |
                   TaxClass of CarClass*real |
                   TaxYear of int*int*real;

abstype CarOrder =  ..... Your representation here...
```

```
with
   local
     .....  Your hidden declarations here...
   in
     exception Error of string;
     .....  Your interface function definitions here...
   end;
end;
```

You are free to choose any internal representation for your datatype as long as you provide the following integrity rules:

- All occurrences of items with same (*Producer,Model,Class,Year*) values should have the same *unit price*.
- No items with same (*Producer,Model,Class,Year,Color*) should be duplicated in the reports or returned values. They should be reported as a single entry with amount summed up. Note that you are free to provide this in the representation or in the interface functions.
- None of the value constraints from the order definitions and tax rule definitions should be violated in the type. You should prevent this in the interface functions by raising exceptions in the requests for insertion of invalid values.
- Order of tax rules is preserved and they are kept in the insertion order, first inserted should be first applied and so on.

You should provide the following interface functions with `CarOrder` type:

`emptyOrder` *`taxdefault`*`: real -> CarOrder`
: Get the default tax rate in percentage (10.0 is 10%) and return an empty order without any entries.

`insertOrder` *`orders (prod,model,year,class,color,amount,price)`*`:`
: `CarOrder -> string*string*int*CarClass*string*int*real -> CarOrder`
Get the existing `CarOrder` typed value *`orders`* and return the new `CarOrder` typed value with the order entry described by the 7-tuple is added to *`orders`*. If a similar entry (first 4 values in the tuple is the same) exists in the *`orders`* new `price` value is considered as the new unit price for all matching entries. Note that entries with different colors may exist but they should have the same price!

`deleteOrder` *`orders (prod,model,year,class,color)`*`:`
: `CarOrder -> string*string*int*CarClass*string -> CarOrder`
Search the 5-tuple in the *`orders`* and return the new order value with the matching entry is deleted. If entry is not found an exception is raised.

`getOrder` *`orders (prod,model,year,class,color)`*`:`
: `CarOrder -> string*string*int*CarClass*string -> int*real*real`
Search the 5-tuple in the *`orders`* and return the amount, unit price and the tax percentage applied for the entry in a triple. If entry is not found an exception is raised.

`orders1 ++ orders2``:`
: `CarOrder*CarOrder -> CarOrder`
Add the entries in *`orders1`* and *`orders2`* and return a new `CarOrder` value where entries come from the addition operation and default tax rate and tax rules come from the *`orders2`*. In the addition process amount fields of the similar entries will be summed up and prices from the *`orders2`* will override. As a result, integrity of the new value should be preserved.

orders1 `**` *orders2*: `CarOrder*CarOrder -> CarOrder`
: Take the intersection of two orders. In the return value order entries will come from the entries only existing in both *orders1* and *orders2*. The amount in the new entry should be the minimum of two amounts and unit price should be the price from the entry in *orders2*. Default tax rate and tax rules should come from *orders2*. `**` operator has a higher precedence than `++`

`totalOfOrder` *orders*: `CarOrder -> real`
: Get the *orders*, calculate and return total cost of the order including taxes

`showOrder` *orders*: `CarOrder -> unit`
: Get the *orders* and produce a report on standart output. Report will have the following format (first line is just a ruler, not a part of the output):

```
0123456789012345678901234567890123456789012345678901234567890123456789012345678
Producer    |Model     |Year|Class    |Color    |Amt|Price    |Tax  |Total
------------+----------+----+---------+---------+---+---------+-----+----------
Ferrari     |F1        |2000|sports   |Red      |  1| 100000.0| 30.2|  112000.0
Tofas       |Sahin     |1984|sedan    |Blue     |  4|   2345.5| 10.0|   10320.2
Tofas       |Sahin     |1984|sedan    |Yellow   |  2|   2345.5| 10.0|    5160.1
------------+----------+----+---------+---------+---+---------+-----+----------
Total                                              5                   127480.3
```

Use following functions to have an aligned string for real, int and string values:

```
fun fmtreal n r = StringCvt.padLeft #" " n
                      (Real.fmt (StringCvt.FIX (SOME 1)) r);
fun fmtint n i = StringCvt.padLeft #" " n (Int.toString i);
fun fmtstring n s = StringCvt.padRight #" " n s;
```

These functions will produce longer strings if the value does not fit into `n` positions. You should print the string with overflows and ignore the shifted layout. No need to mention, you should put these function definitions in the hidden part in your declaration block. Entries should be sorted in the ascending order according to producer, model, class, year and color. Primary ordering will be done according to producer name, for entries with same producer name secondary ordering will be done according to model name, then according to class name and so on... string comparisons will be case sensitive and based on `<` operator. For `CarClass` corressponding string names will be significant in the comparison.

`insertTaxRule` *orders taxrule*: `CarOrder -> TaxRule -> CarOrder`
: Get a `CarOrder` typed value *order* and return a new order with *taxrule* is inserted at the end of the tax rules list. There can be multiple tax rules with exactly same values. In this case rule will be applied multiple times in the insertion order as if they were distinct rules.

`deleteTaxRule` *orders taxrule*: `CarOrder -> TaxRule -> CarOrder`
: Get a `CarOrder` type value *order* and return a new order with all matching entries for *taxrule* are deleted from the tax rule list of *order*. If no matching entry is found and exception is raised.

You have an exception definition '`exception Error of string`' in interface part and you should produce the following values in the exceptions:

`Error "Entry is not found in get or delete"`	by `getOrder` or `deleteOrder`
`Error "Invalid numeric value in the order"`	by `insertOrder` or `emptyOrder`

`Error "Tax rule is not found"`	by `deleteTaxRule`
`Error "Invalid range or value in tax rule"`	by `insertTaxRule`

All string operations are case sensitive. Examples will be provided before the weekend.