Fall 2013
Programming Languages
Homework 3

- This homework is an ML programming assignment. You should use Standard ML of New Jersey for the programming portion of the assignment. A link is available on NYU Classes.
- Due on Tuesday, November 12, 2013, 3:00 PM Eastern Daylight time. Submit a single ML solution file for as an attachment.
- Do not use imperative features such as assignment `:=`, references (keyword `ref`), or any mutable data structure, such as `Array`.
- You may consult ML references (e.g. textbooks, Internet, etc.) and with other students to improve your understanding of the ML programming language. You may exchange ML code with other students, provided it has no relationship to this assignment. You may use the NYU Classes forums for such communication.
- You **may not**, under any circumstance, collaborate with students or use any reference material (in any form) in solving any homework problems. All of your homework solutions including algorithmic details, specific approaches used, actual ML code, etc., **must** be yours alone.
- Assignments will be inspected both manually and automatically for plagiarism. Violations of the academic dishonesty policy are subject to disciplinary action, regardless of when the academic dishonesty is discovered.
- Consult the course page for helpful resources on ML. Please also see `http://www.smlnj.org/doc/errors.html` for information on common ML errors.
- Please regularly check the Homework 3 forum on NYU Classes for answers to questions, updates and/or clarifications regarding this homework assignment.

1 Hotel Reservation Library

A hotel needs a library which their software staff can use for managing room reservations. The library shall consist of a *signature* and corresponding *structure*. The signature will expose an API of public-facing types, functions, and exceptions related to managing hotel reservations. The structure will contain the internal library implementation, which may specify additional functions, type definitions, or other detail not visible to the user of the library.

1.1 Background

A hotel room for the system in question can have three (3) possible configurations: 2 double beds, 1 queen bed, or 1 king bed.

The length of a stay in the hotel business is measured in *nights*, commencing from the first night. For example, a guest staying from November 17 - 19 is staying for 2 nights: the night of November 17th and 18th.

Customers can book a *stay*—one room of a particular room configuration, starting on a particular night for one or more consecutive nights. The *inventory* for a particular room configuration on a particular night is the number of rooms in the hotel with that configuration, less the number of guests staying in those rooms on that night. Whenever a room is reserved, the inventory for the reserved configuration decreases by 1 for all nights of the stay. Whenever a reservation for the same room is canceled, inventory increases by 1 for all nights of the stay.

When a guest makes a reservation, a *reservation record* consisting of a unique ID, the guest's first name, last name, date of check-in number of nights, number of occupants, and room configuration should be added to the reservation system. Reservations can be confirmed or denied at the time the reservation is made. The reservation process should check that the inventory is available for all nights of the stay prior to confirming the reservation. Otherwise the reservation system should deny the reservation.

Reservations can be canceled too. For simplicity, the library should delete the reservation from the system if it is canceled. Reservations can be canceled at any time, including in the middle of a stay.

So that the size of the reservation database does not grow indefinitely, there must exist a mechanism to remove reservations for all stays that *end* before a specified date.

Miscellaneous notes: for this exercise, we're not concerned about room prices, special requests, upgrades, promotional rates, or any other real-life detail beyond what is mentioned above. For convenience, you may express dates as non-negative integers (e.g. the number of days elapsed since some known point in time). In other words, you don't need to parse date strings or concern yourself with specialized date types.

1.2 Implementation

Begin the implementation by writing a signature called `HOTELRESERVATIONS`. The corresponding structure should be called `HotelReservations`. Remember that it is ML convention that type names (including datatypes) are typically all lowercase, whereas data constructors usually begin with a capital letter.

The hotel reservation system should be comprised of two main data structures: the first is a *reservation record* whose type should be named `resrecord`, a record holding the details of a single stay. You have the freedom to choose the exact representation (i.e., tuples, records, or datatypes). The second is a *reservation system* whose type should be named `ressys`. This is a data structure representing all information concerning the hotel, including room quantities and reservations records.

Additionally, define a `datatype` named `roomconfig` to represent a room configuration, with one data constructor per configuration. The reservation record described above will therefore contain a value of type `roomconfig`.

Visibility: the signature should expose the full details of types `resrecord` and `roomconfig`, since the user of the library will need to use these definitions in order to know what information to pass when creating room reservations. This is accomplished by placing the details in both the signature and structure. On the other hand, the signature should expose the existence of type `ressys`, but not the implementation details. To do this, `type ressys` should be specified in the signature, but the structure should contain the full definition of this type. The implementation details contained within the structure remain hidden from the library user.

When writing the library functions (described below), you may find the need to define "helper" functions which perform some aspect of the computation necessary for a particular function. In these cases, consider whether the helper function is used by only one function or several other functions. If only one, consider defining it inside a `let` clause to make it visible to only the function requiring the helper. If you write a helper function that can be used more generally, place it in your structure so that it will be available to the rest of the functions in the structure, but not to the user.

Anywhere that the requirements require an exception to be raised, the exception need not be caught since doing so is the responsibility of the library user.

1.3 Requirements

Your library must specifically contain *at least* the following functions with the specified signatures. The types as shown above must be publicly exposed as part of the signature, but the implementation detail should not:

- `empty : int -> int -> int -> ressys`, given the quantity of available hotel rooms for each configuration—2 double beds, 1 queen bed, and 1 king bed, respectively—will evaluate to an empty reservation system (a system with no reservation records).

- `reserve : ressys -> resrecord -> ressys`, given a reservation system and a reservation record, creates a new reservation for one room and evaluates to the new reservation system. Raises an exception if there isn't enough room inventory to satisfy the reservation, or if there already exists a reservation with the given unique ID.
- `cancel : ressys -> int -> ressys`, given a reservation system and the unique ID of a reservation record, cancels an existing reservation and evaluates to the new reservation system. Raises an exception if the reservation record identified by the unique ID doesn't exist in the system.
- `getInventory : ressys -> roomconfig -> int -> int`, given a reservation system, a room configuration, and a date, evaluates to the number of rooms available to be reserved for the specified configuration for the given date.
- `getInventorySpan : ressys -> roomconfig -> int -> int -> bool`, given a reservation system, room, date, and number of nights, evaluates to `true` if the room inventory is available for all nights, or `false` otherwise.
- `completedStays : ressys -> int -> int`, given a reservation system and a date, returns the number of completed stays as of the date. (A stay is completed if all nights of the stay are before the specified date.)
- `removeCompletedStays : ressys -> int -> ressys`, given a reservation system and a date, evaluates to a new reservation system that does not include any fully completed stays. *Hint: consider using* `filter` *from slide 16 of the ML lecture.*

I recommended that you implement the above functionality before proceeding to the next set of requirements.

1.4 More Requirements

1. The first three parameters to the `empty` function above (the quantity of each room configuration) should be fixed for a particular hotel, since the number of rooms in the hotel is fixed. Build these quantities into the definition of the structure by writing a *functor* named `MakeHotel` which, given a structure containing the quantity of each room configuration, creates the same structure as `HotelReservations`. In doing so, be sure to rewrite the signature of `empty` to remove the first 3 parameters.

   You might define the following signature:

   ```
   signature ROOMDETAIL =
     sig
       val doubleAvailable : int
       val kingAvailable : int
       val queenAvailable : int
     end;
   ```

Your functor might then look something like:

```
functor MakeHotel (structure Q : ROOMDETAIL) :
sig
  (* What was previously contained in the
     HOTELRESERVATIONS signature goes here *)
  ...
   val empty : ressys;  (* no more params *)
  ...
end = struct
  (* What was previously contained in the
     HotelReservations structure goes here *)
  ...
end;
```

Later when invoking the functor, you might write:

```
structure HiltonRoomDetail :
  sig
    val doubleAvailable = 5
    val kingAvailable = 4
    val queenAvailable = 3
  end;

structure MariottRoomDetail :
  sig
    val doubleAvailable = 3
    val kingAvailable = 6
    val queenAvailable = 4
  end;

structure HiltonHotelReservations =
         MakeHotel(structure Q = HiltonRoomDetail);

structure MarriottHotelReservations =
         MakeHotel(structure Q = MarriottRoomDetail);
```

2. Modify the `resrecord` type so that each reservation record also includes the number of guests staying in the room.

3. Reservation systems place restrictions on reservations. For example, the hotel may require a minimum number of nights per stay or an occupancy limit on each room. To the `ROOMDETAIL` signature, add the additional components

```
val minnights : int OPTION
val occupancyLimit : int
```

representing an optional minimum night requirement and the room occupancy limit. Modify any structures implementing `ROOMDETAIL` to add these new values.

Define a predicate in the `struct` part of the `MakeHotel` functor, whose signature is `restrictions : resrecord -> bool`. Enforce the minimum night policy and the maximum occupancy restriction in this function, which should evaluate to `true` if all restrictions are satisfied, or `false` otherwise. Modify your definition of `reserve` to call `restrictions`, and raise exceptions in the event of a failure.

4. Add `guestQuantity : ressys -> int -> int` to the signature part of the functor. Given a reservation system and a date, function `guestQuantity` evaluates to the total number of guests staying in the hotel on the specified night. Implement the function in the `struct` part of the functor.

5. Make your own custom addition or improvement to the reservation system. Explain what the feature or improvement is.

1.5 Deliverables

All ML code should be submitted in a single ML file. You only need to turn in the fully completed project—not any of the intermediary steps. Comment your code thoroughly. For each function, write the purpose of the function as well as the expected input and output.

Write a test bed, to appear at the bottom of your ML file, that performs the following actions:

1. Instantiate a structure adhering to the ROOMDETAIL signature. In doing so, specify a room quantity for each configuration, an occupancy limit of 4 persons per room and a minimum night requirement of 2 nights.

2. Instantiate at least one hotel reservation structure using the functor, passing the structure from the step above.

3. Add at least 5 reservations.

4. Demonstrate the unsuccessful creation of a reservation due to lack of inventory for at least one room configuration.

5. Demonstrate the unsuccessful creation of a reservation due to either the room occupancy being exceeded or minimum night requirement not being met.

6. Cancel at least 2 reservations.

7. Query the room inventory for a particular configuration, for a particular date.

8. Demonstrate `removeCompletedStays` and `completedStays`. There should be at least 2 completed stays in the system prior to the demonstration. After old reservations are removed, function `completedStays` should evaluate to 0.

9. Demonstrate the `guestQuantity` function for a particular night.

10. Demonstrate the new feature or improvement that you implemented above.

You will be graded primarily on whether your code meets the above requirements, functions properly, and is well-commented. Do not get hung up on runtime performance or style issues. This is assumed to be everyone's first exposure to programming in a functional language.

1.6 Advice

The main purpose of this homework assignment is to force you to "think functionally" about solving problems. Most programmers who have worked primarily with imperative languages in the past tend to be strongly biased toward imperative thinking, which in many cases can serve as an obstacle to getting the most out of functional languages.

Imperative programmers tend to ask the question, "how must I change the state of objects to arrive at the solution to my problem?" whereas functional programmers will ask "what expressions must I create to arrive at the solution to my problem?" A sorting algorithm serves as a good example. An imperative programmer will *modify* the contents of an unsorted list to arrive at the same physical list as before, but whose contents are now sorted. A functional programmer will use an unsorted input list to *create* a new sorted list—in doing so, many new shorter lists may be created along the way.

One notable distinction between functional and imperative programs is that the functional program tends to be syntactically shorter and simpler-looking, yet generates the same end result as a much larger, more sophisticated imperative program. I recommend approaching the assignment in the most simplistic way possible. Represent collections using lists. Use pattern matching or the `hd` and `tl` functions to access the head and tail of lists. Use recursion to iterate through the lists, using the body of the function to test for various conditions that you are interested in. Look at and understand the ML examples from the lecture.

My advice is to get your types defined and your functions working individually first, defining everything at the global level, and then place everything into signatures/structures later after everything is working. Spend the first day or two getting the hang of functional programming by writing some simple programs. Most students find that after the initial learning curve has been overcome, the assignment goes much more smoothly.

Whatever you do, **get started now**. Do not wait until next week or the last few days to start.