MIT
6.031: Software Construction
Max Goldman and Prof. Armando Solar-Lezama revised Sunday 5th May, 2019, 23:39

Quiz 2 (May 6, 2019)

Your name: _____

Your Kerberos username: _____

You have 50 minutes to complete this quiz. It contains 17 pages (including this page) for a total of 100 points.

The quiz is closed-book and closed-notes, but you are allowed one two-sided page of notes.

Please check your copy to make sure that it is complete before you start. Turn in all pages, together, when you finish. Before you begin, write your Kerberos username on the top of every page.

Please write neatly. **No credit will be given if we cannot read what you write.**

For questions which require you to choose your answer(s) from a list, do so clearly and unambiguously by circling the letter(s) or entire answer(s). Do not use check marks, underlines, or other annotations – they will not be graded.

Good luck!

Problem	Points
1: Concurrency	20
2: Little Languages	30
3: Parsing and Grammar	20
4: Map Reduce Filter	30
Total	100

The first three problems in this quiz are based on the same abstract datatype as Quiz 1. Below is a definition from the previous quiz to refresh your memory. The code can be found at the end of the quiz. Feel free to detach the code pages starting at page 13.

A *thermostat program* describes the settings of a simple temperature control system that has a *mode* and an association between blocks of time and the *goal temperatures* to try to maintain during those times.

The system mode is either *heat* or *cool*. We are ignoring other possible common settings like *auto* or *off*.

The granularity of our thermostat programs is *30-minute blocks starting on the hour and half-hour*: 12 midnight, 12:30 am, 1:00 am, 1:30 am, *etc*. Each 30-minute block has an associated *goal temperature* which the system will try to maintain by using one of heating or cooling, depending on the mode.

For example, here is a possible thermostat program for a home during winter, where the times indicate the start of each 30-minute block:

Mode: heat

12 midnight ... 6:00 am	**65** °**F** (overnight temperature, sleeping)
6:30 am ... 8:30 am	**68** °**F** (warmer for waking up, breakfast)
9:00 am ... 12 noon ... 5:00 pm	**62** °**F** (cooler while everyone is at work/school)
5:30 pm ... 10:00 pm	**68** °**F** (warmer for dinner, going to sleep)
10:30 pm ... 11:30 pm	**65** °**F** (overnight temperature, sleeping)

Problem 1 (Concurrency) (**20 points**).
Ben BitTwiddle decides to take the thermostat program from the last quiz and connect it to the internet. What could go wrong! Ben does this by creating a `ThermostatServer` class that will receive requests from clients and update the thermostat based on those requests. Ben paid attention during the sockets class, so made sure to have separate threads for each client to allow them to enter requests concurrently. Unfortunately, Ben was sick during the thread safety class, so the code will have some problems during execution because the `ThermostatProgram` class was not designed to be threadsafe.

(a) Suppose we change the `tempRules` initialization to

```
this.tempRules = Collections.synchronizedNavigableMap(new TreeMap<>());
```

This helps with thread safety but it is not enough. Explain why this helps, but how `ThermostatProgram` could fail when its methods are called concurrently (for example, by showing a bad interleaving).

(b) Frustrated, Ben decides to avoid trouble by using a lock. Specifically, Ben changes the declaration of `handle` to

```
private synchronized void handle(Socket socket) throws IOException
```

Does this fix the thread safety problems? Yes/No and explain why. Make explicit any assumptions you are making about other parts of the code.

YES / NO

(c) Describe a situation that could prevent the code from part (b) with the **`synchronized`** `handle` from making progress for an indefinite amount of time.

(d) Does the situation you identified in the previous question correspond to a deadlock? Why or why not?

Problem 2 (Little Languages) (**30 points**).
Programming the thermostat a half an hour interval at a time can get messy, so instead, we are going to develop a little language to specify more complex time intervals.

(a) The language is defined around a datatype named `TimeSet` that represents a set of times. There are two variants of `TimeSet`: `Range` takes two points in time and returns a `TimeSet` representing that time interval. For example, `TimeSet.range(Time.time(5, 30), Time.time(6,0))` represents the span of time starting at 5:30 AM and ending at 6AM.
The second variant `Union` represents the union of two `TimeSet`s. (See the definition on page 16.)

Write a datatype definition for this little language:

(b) The `TimeSet` interface requires a `hasTime` method that determines whether a given time is part of the `TimeSet`. Write an implementation of the `hasTime` method of `Union`.

```
public boolean hasTime(Time t) {
```

```
}
```

(c) Ben wants to make it easy to add new functions over the `TimeSet` datatype, so defines a visitor interface as follows.

```
public interface TimeSetVisitor<R> {
    public R on(Range r);
    public R on(Union u);
}
```

Using this interface, it is possible to define, for example, a `PrintToString` visitor.

```
public class PrintToString implements TimeSetVisitor<String> {
    /**
     * @return produces a string representing the Range r.
     */
    public String on(Range r) { return /* Code to print a range */ }
    /**
     * @return produces a string representing the Union u.
     */
    public String on(Union u) { return /* Code to print a Union */ }
}
```

Ben wants to use the visitor as follows:

```
TimeSet times = TimeSet.union(TimeSet.range(Time.time(5,30), Time.time(6,0)),
                              TimeSet.range(Time.time(8,30), Time.time(10,0)));
PrintToString print = new PrintToString();
print.on(times);
```

Unfortunately, the code above will not work. Explain why not.

(d) Explain what changes need to be made to the code in order to be able to use the `TimeSetVisitor` correctly and why those changes are necessary.

(e) Assuming the changes from your answer to part (d), complete the body of `String on(Union u)` inside `PrintToString`.

We expect, for example, the `times` object earlier to render as "(5:30 , 6:00) U (8:30 , 10:00)". You may assume a correct implementation of `on(Range r)`.

```
public String on(Union u) {
```

```
}
```

Problem 3 (Parsing and Grammar) (**20 points**)**.**
Continuing with the example from the previous problem, in order to make it even easier to program the thermostat, Ben decides to implement a parser to generate `TimeSet`s from text files. The grammar Ben writes to do this is shown below:

```
Time ::=  number ':' number ;
Range ::= "("  Time  ", "  Time  ")";
TimeSetExpression ::= (Range)*;
number ::= [0-9]+;
```

The format that Ben has in mind will represent Time as "hour:minute" using a 24 hour format, so 3:30PM would be "15:30". The `Range` represents a time interval between two times, and a `TimeSetExpression` corresponds to the `TimeSet` datatype and represents the Union of multiple ranges.

(a) Show how to change the grammar above to make optional both the minutes (so a user can write 12 instead of 12:00) and the second time (so a user can write (15:00) instead of the interval with a single half hour fragment (15:00, 15:30)).

(b) Suppose we want to extend the language from the previous part (with support for optional minutes and end of the range) to include commands for actually setting the temperature for a particular TimeSet. A proposal for doing this is to extend the grammar as follows:

```
Command ::= '('  targetTemperature  ')'  TimeSetExpression ;
Program ::= (Command)*;
targetTemperature ::= number;
```

Is this a good proposal? Explain why or why not and whether your answer would change if you did this in terms of the original grammar (instead of the modified grammar from the previous question).

(c) How would you fix the `Command` definition to avoid any problems you identified in the previous question?

Problem 4 (Map Reduce Filter) (**30 points**).
And now, a problem that has nothing to do with thermostats. The questions that follow assume you have a `User` class like the one shown below, which has a `getLiked()` method which returns a list of all the Users this user has liked. It also has an `equals()` method to check whether two users are the same user.

```
public class User {
    /**
     * @return users this user has liked; liking is not necessarily mutual,
     * so users I have liked may not like me.
     */
    public List<User> getLiked() { ... }
    @Override
    public boolean equals(Object o) { ... }
}
```

Answer the following questions using only the `map`, `flatMap`, `reduce`, `filter` we studied in class. You are also allowed to use the `stream()` method to get the stream from a list, and the `count()` method to count the number of elements in a stream. You can also use any methods in the `User` class, but **no additional methods**. You also cannot use loops. Type signatures for these functions are given below for your convenience, as instance methods of `Stream<A>`:

```
Stream<A>::map : (A → B) → Stream<B>
Stream<A>::flatMap : (A → Stream<B>) → Stream<B>
Stream<A>::filter : (A → Boolean) → Stream<A>
Stream<A>::reduce : (A × (A × A → A) ) → A
Stream<A>::reduce : (B × (B × A → B) × (B × B → B) ) → B
```

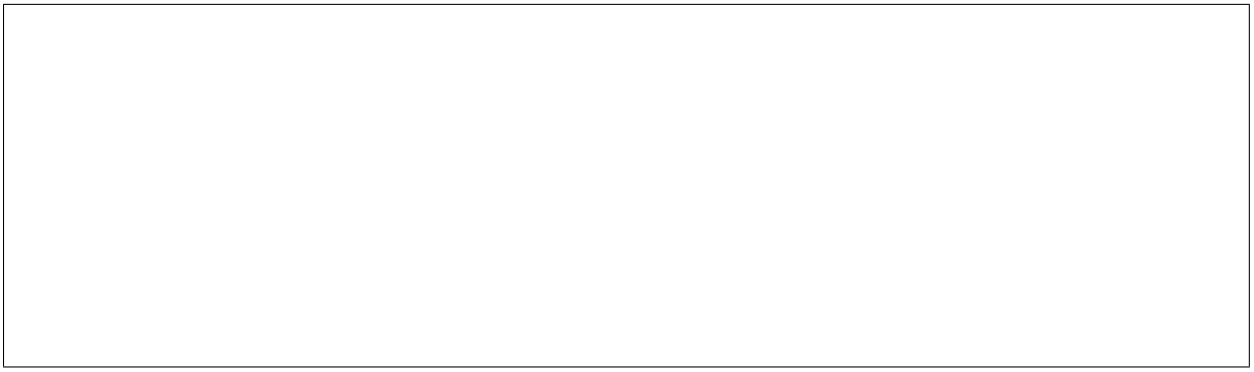
(a) Given a list of users and a user bob, return a stream containing all the users that have liked bob (*i.e.* that have bob in their `getLiked` list).

```
Stream<User> likeBob(List<User> users, User bob) {
```

```
}
```

(b) Given a list of users, return a stream of all their liked users (it is ok for the stream to have duplicates).

```
Stream<User> allLiked(List<User> users) {
```



```
}
```

(c) Given a list of users, return a stream that excludes all the users that appear more than once (*i.e.* given a list [u1, u2, u1, u3], you want a stream representing the list [u2, u3], since u1 appears more than once, so it gets excluded.) Hint: That `count()` function mentioned earlier may come in handy here.

```
Stream<User> excludeMultiples(List<User> users) {
```

```
}
```

SFB
ETU
RFC

You may detach this and all subsequent pages. Write your username at the top, and hand in all pages when you leave.

```
1  /** Mutable thermostat program that is defined using rules. */
2  public class ThermostatProgram {
3      public enum Mode { COOL, HEAT }

4      private Mode mode;
5      private final NavigableMap<Integer,Integer> tempRules;

6      public ThermostatProgram(Mode mode) {
7          this.mode = mode;
8          this.tempRules = new TreeMap<>();
       }

9      /*
10      * Excluded some methods from the original ThermostatProgram
11      * that are not relevant for this quiz.
12      */

13     /**
14      * Modify this program to add a rule starting at the given time (or replacing
15      * the rule starting at the given time, if any) with goal temperature 'temp'.
16      * The latest-time rule carries over through midnight to the next day.
17      * @param hour must be 0 <= hour < 24
18      * @param minute must be 0 or 30
19      * @param temp goal temperature in degrees Fahrenheit
20      */
21     public void addRule(int hour, int minute, int temp) {
22         tempRules.put(hour * 2 + (minute < 30 ? 0 : 1), temp);
       }

23     /**
24      * Modify this program to remove the rule (if any) that currently determines
25      * the goal temperature for the given time.
26      * The latest-time rule carries over through midnight to the next day.
27      * @param hour must be 0 <= hour < 24
28      * @param minute must be 0 or 30
29      */
30     public void removeRule(int hour, int minute) {
31         if (tempRules.isEmpty()) {
32             return;
           }
33         Integer ruleTime = tempRules.floorKey(hour * 2 + (minute < 30 ? 0 : 1));
34         if (ruleTime != null) {
35             tempRules.remove(ruleTime);
36             return;
           }
37         tempRules.remove(tempRules.lastKey());
       }
   }
```

```
1  /* ThermostatServer for Problem 1 */

2  public class ThermostatServer {
3      public static final int PORT = 4949;
4      public enum RequestKind { ADD, REMOVE, SWITCH }

5      /** Represents a request sent by the client. */
6      private static class Request {
7          /* Excluded some code not relevant to the quiz */
8          RequestKind getKind() { return kind; }
9          int getHour() { return hour; }
10         int getMin() { return minute; }
11         int getTemp() { return temp; }
       }

12     private final ServerSocket serverSocket;
13     private ThermostatProgram thermostat;

14     /** Make a ThermostatServer that listens for connections on port.
15      *  @param port port number, requires 0 <= port <= 65535
16      *  @throws IOException if there is an error listening on port */
17     public ThermostatServer(int port, ThermostatProgram thermostat) throws IOException {
18         serverSocket = new ServerSocket(port);
19         this.thermostat = thermostat;
       }

20     /** Run the server, listening for connections and handling them.
21      *  @throws IOException if the main server socket is broken */
22     public void serve() throws IOException {
23         System.err.println("serving");
24         while (true) {
25             // block until a client connects
26             final Socket socket = serverSocket.accept();
27             new Thread(() -> {
28                 try {
29                     handle(socket);
30                 } catch (IOException ioe) {
31                     ioe.printStackTrace(); // but don't terminate serve()
32                 } finally {
33                     try {
34                         socket.close();
35                         System.err.println("Connection closed");
36                     } catch (IOException ioe) {
37                         ioe.printStackTrace(); // and still don't terminate serve()
                       }
                   }
38             }).start();
           }
39     }
       // continues on next page
```

```
    // ThermostatServer, continued

40  /** Function waits for a request to be available from
41   *  a socket, reads it and then closes the connection.
42   *  @param socket from which the request will be read.
43   *  @return a Request object representing the request.
44   */
45  Request parseRequest(Socket socket) {
46      ...
    }


47  /** Reads a single request from the client and
48   *  ends the connection.
49   *  @param socket socket where client is connected
50   *  @throws IOException if connection encounters an error
51   */
52  private void handle(Socket socket) throws IOException {
53      Request req = parseRequest(socket);
54      switch(req.getKind()) {
55      case ADD: {
56          thermostat.addRule(req.getHour(), req.getMin(), req.getTemp());
57          return;
        }
58      case REMOVE: {
59          thermostat.removeRule(req.getHour(), req.getMin());
60          return;
        }
61      case SWITCH: {
62          thermostat.switchMode();
63          return;
        }
        }
    }
}
```

```
/* TimeSet interface and classes for Problems 2 and 3. */

class Time {
    final int hour;
    final int min;
    Time(int hour, int min) {
        this.hour = hour;
        this.min = min;
    }
    /**
     * Factory method to construct a new Time object.
     */
    public static Time time(int hour, int min) {
        return new Time(hour, min);
    }
}

/**
 * Set of times in half hour intervals.
 */
public interface TimeSet {
    /**
     * Represents a range of times in half hour intervals
     * starting at time start and ending at time end.
     * @param start Start of the time interval.
     * @param end End of the time interval.
     */
    public static TimeSet range(Time start, Time end) {
        return new Range(start, end);
    }
    /**
     * @return a TimeSet representing the union of left and right TimeSets.
     */
    public static TimeSet union(TimeSet left, TimeSet right) {
        return new Union(left, right);
    }
    /**
     * @return boolean value that determines whether time t belongs to a TimeSet
     */
    public boolean hasTime(Time t);
}
```

// continues on next page

```
// TimeSet interface and classes, continued

class Range implements TimeSet {
    Time start;
    Time end;
    Range(Time start, Time end) {
        this.start = start;
        this.end = end;
    }
    public boolean hasTime(Time t) {
        ...
    }
}

class Union implements TimeSet {
    TimeSet left;
    TimeSet right;
    Union(TimeSet left, TimeSet right) {
        this.left = left;
        this.right = right;
    }
    public TimeSet getLeft() {
        return left;
    }
    public TimeSet getRight() {
        return right;
    }
    public boolean hasTime(Time t) {
        ...
    }
}
```