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!

<table>
<thead>
<tr>
<th>Problem</th>
<th>Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: Concurrency</td>
<td>20</td>
</tr>
<tr>
<td>2: Little Languages</td>
<td>30</td>
</tr>
<tr>
<td>3: Parsing and Grammar</td>
<td>20</td>
</tr>
<tr>
<td>4: Map Reduce Filter</td>
<td>30</td>
</tr>
<tr>
<td>Total</td>
<td>100</td>
</tr>
</tbody>
</table>
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
6:30 am
...
8:30 am
9:00 am
...
12 noon
...
5:00 pm
5:30 pm
...
10:00 pm
10:30 pm
...
11:30 pm
```

65 °F (overnight temperature, sleeping)

68 °F (warmer for waking up, breakfast)

62 °F (cooler while everyone is at work/school)

68 °F (warmer for dinner, going to sleep)

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

```java
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

```java
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
(e) 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 6 AM.
The second variant Union represents the union of two TimeSets. (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) {

}
(e) Ben wants to make it easy to add new functions over the \texttt{TimeSet} datatype, so defines a visitor interface as follows.

\begin{verbatim}
public interface TimeSetVisitor<R> {
    public R on(Range r);
    public R on(Union u);
}
\end{verbatim}

Using this interface, it is possible to define, for example, a \texttt{PrintToString} visitor.

\begin{verbatim}
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 */ }
}
\end{verbatim}

Ben wants to use the visitor as follows:

\begin{verbatim}
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);
\end{verbatim}

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)`.

```java
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 TimeSets 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).

(e) 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.

```java
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).

```java
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).

```java
Stream<User> allLiked(List<User> users) {
    // Your implementation here
}
```

(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.

```java
Stream<User> excludeMultiples(List<User> users) {
    // Your implementation here
}
```
Your Kerberos username:____________________

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.

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

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

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

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

    /**
     * Modify this program to add a rule starting at the given time (or replacing
     * the rule starting at the given time, if any) with goal temperature 'temp'.
     * The latest-time rule carries over through midnight to the next day.
     * @param hour must be 0 <= hour < 24
     * @param minute must be 0 or 30
     * @param temp goal temperature in degrees Fahrenheit
     */

    public void addRule(int hour, int minute, int temp) {
        tempRules.put(hour * 2 + (minute < 30 ? 0 : 1), temp);
    }

    /**
     * Modify this program to remove the rule (if any) that currently determines
     * the goal temperature for the given time.
     * The latest-time rule carries over through midnight to the next day.
     * @param hour must be 0 <= hour < 24
     * @param minute must be 0 or 30
     */

    public void removeRule(int hour, int minute) {
        if (tempRules.isEmpty()) {
            return;
        }
        Integer ruleTime = tempRules.floorKey(hour * 2 + (minute < 30 ? 0 : 1));
        if (ruleTime != null) {
            tempRules.remove(ruleTime);
            return;
        }
        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
6    /** Represents a request sent by the client. */
7    private static class Request {
8        /* Excluded some code not relevant to the quiz */
9        RequestKind getKind() { return kind; }
10       int getHour() { return hour; }
11       int getMin() { return minute; }
12       int getTemp() { return temp; }
13    }
14
15    private final ServerSocket serverSocket;
16    private ThermostatProgram thermostat;
17
18    /** Make a ThermostatServer that listens for connections on port. */
19    * @param port port number, requires 0 <= port <= 65535
20    * @throws IOException if there is an error listening on port */
21    public ThermostatServer(int port, ThermostatProgram thermostat) throws IOException {
22        serverSocket = new ServerSocket(port);
23        this.thermostat = thermostat;
24    }
25
26    /** Run the server, listening for connections and handling them. */
27    * @throws IOException if the main server socket is broken */
28    public void serve() throws IOException {
29        System.err.println("serving");
30        while (true) {
31            final Socket socket = serverSocket.accept();
32            new Thread() -> {
33                try {
34                    handle(socket);
35                } catch (IOException ioe) {
36                    ioe.printStackTrace(); // but don’t terminate serve()
37                } finally {
38                    try {
39                        socket.close();
40                        System.err.println("Connection closed");
41                    } catch (IOException ioe) {
42                        ioe.printStackTrace(); // and still don’t terminate serve()
43                    }
44                }
45            ).start();
46        }
47    }
48
49    // continues on next page
// ThermostatServer, continued

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

/** Reads a single request from the client and ends the connection. * @param socket socket where client is connected * @throws IOException if connection encounters an error */
private void handle(Socket socket) throws IOException {
    Request req = parseRequest(socket);
    switch(req.getKind()) {
        case ADD:
            thermostat.addRule(req.getHour(), req.getMin(), req.getTemp());
            return;
        case REMOVE:
            thermostat.removeRule(req.getHour(), req.getMin());
            return;
        case SWITCH:
            thermostat.switchMode();
            return;
    }
}
class Time {
    final int hour;
    final int min;
    Time(int hour, int min) {
        this.hour = hour;
        this.min = min;
    }
}

public static Time time(int hour, int min) {
    return new Time(hour, min);
}

public interface TimeSet {
    public static TimeSet range(Time start, Time end) {
        return new Range(start, end);
    }
    public static TimeSet union(TimeSet left, TimeSet right) {
        return new Union(left, right);
    }
    public boolean hasTime(Time t);
}

// continues on next page
// TimeSet interface and classes, continued

```java
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) {
        ...
    }
}
```