You have 50 minutes to complete this quiz. It contains 9 pages (including this one). The last page is blank for scratch space.

Before you begin, write your Athena username on the top left of every page.

Answer the questions in the spaces provided on the question sheets.

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

For questions which require you to circle your answer(s), do so clearly and unambiguously. Circle \( A \) the letter or \( B \) the entire answer. Do not use check marks, underlines, or other annotations – they will not be graded.

Good luck!

Name: ________________________________________________

Athena username: ______________________________________

<table>
<thead>
<tr>
<th>Page</th>
<th>Points</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>100</td>
<td></td>
</tr>
</tbody>
</table>
Short Multiple Choice

1. Given the regular expression: \([ac]_+_[123]^*\), circle all the strings that match:
   A. _223  B. b_1  C. ccca32  D. cac_
   (4pts)

2. Given the string _foo3i, circle all the regular expressions that match the string:
   A. [a-z0-9]*  B. [d-p_]+r*b*[h-k0-9]*[a-f]*  C. _foo3i  D. [1-5_a-z]+  
   (4pts)

For the following questions, circle True or False.

3. For an abstract data type, if you can prove the representation invariant is true after the constructor creates a new object, and all mutators preserve the representation invariant, then that means that the RI is always true.
   A. True  B. False  
   (2pts)

4. The visitor pattern is appropriate to use on an immutable data type when it is likely clients will create new operations of their own design on the data type.
   A. True  B. False  
   (2pts)

5. It is bad practice to write tests that test specific code paths in your methods.
   A. True  B. False  
   (2pts)

6. For a well-designed abstract data type, every value in the representation value space represents a value in the abstract value space.
   A. True  B. False  
   (2pts)

7. The abstraction function (AF) maps values in the abstract value space to the representation in the representation value space.
   A. True  B. False  
   (2pts)

Alice writes the following code:

```java
public static int gcd(int a, int b) {
    if (a > b) {
        return gcd(a-b, b);
    } else if (b > a) {
        return gcd(a, b-a);
    }
    return a;
}
```

Bob writes the following test:

```java
@Test public void gcdTest() {
    assertEquals(6, gcd(24, 54));
}
```

The test passes!

For the following questions, circle True or False.

8. Alice should write a > 0 in the precondition of gcd
   A. True  B. False  
   (2pts)

9. Alice should write b > 0 in the precondition of gcd
   A. True  B. False  
   (2pts)

10. Alice should write gcd(a, b) > 0 in the precondition of gcd
    A. True  B. False  
    (2pts)
11. Alice should write $a \& b$ are integers in the precondition of $\gcd$  
   A. True  B. False  

12. Bob should use $\text{assert}$ instead of $\text{assertEquals}$ in his tests  
   A. True  B. False  

13. If Alice adds $a > 0$ to the precondition, Bob should test negative values of $a$  
   A. True  B. False  

14. If Alice does not add $a > 0$ to the precondition, Bob should test negative values of $a$  
   A. True  B. False  

**Testing**

Given the following specification:

```java
/**
 * Return keys of "counts" that map to values above "cutoff".
 */
public static Set<String> findAbove(Map<String, Integer> counts, int cutoff)
```

Suppose our test suite contains:

1. { "a" => 5, "b" => 6 }, 4  
2. { "a" => 5, "b" => 6 }, 5  
3. { "a" => 5, "b" => 6 }, 6  

1. If you can only add 3 additional inputs to the test suite, which of the inputs below would you select?  
   Circle exactly three:
   A. { "a" => 5, "b" => 6 }, null  
   B. { "a" => 5, "b" => 6 }, Integer.MAX_VALUE  
   C. { "a" => 4, "b" => 7 }, 1  
   D. { "a" => 4, "b" => 7 }, 3  
   E. { "a" => 4, "b" => 7 }, 5  
   F. { "a" => 4, "b" => 7 }, 7  
   G. { "a" => 4, "b" => 7 }, 9  


The Immutable Mutable List

Ben Bitdiddle is tired of listening to his coworkers argue over whether they should use a mutable or immutable list implementation. “Fine,” he says, “I’ll write a list implementation that’s both mutable and immutable! Everyone can use it the way they want, and our code will just work.” He writes the code on the next page (which compiles without error).

Reference Ben’s code on the next page to answer these questions:

1. Write out the datatype definition for ImMuList. (5pts)

   ImMuList =

2. For each of the following methods of ImMuCons, classify it as a creator/constructor, producer, mutator, or observer. (6pts)

   i. ImMuCons.isEmpty() A. creator/constructor B. producer C. mutator D. observer
   ii. ImMuCons.prepend() A. creator/constructor B. producer C. mutator D. observer
   iii. ImMuCons.get() A. creator/constructor B. producer C. mutator D. observer

3. Does Ben’s implementation have representation exposure? Fill in the sentence with why or why not. (5pts)

   A. No, because ________________________________.
   B. Yes, because ________________________________.

Suppose Ben has a fit of SW engineering conscience and decides to write specifications for his methods. Here’s the specification for ImMuList.get():

```java
/**
 * Gets the i\(^\text{th}\) element.
 *
 * Given an index i, this method returns the element at that index.
 *
 * @param i index of element to return
 * @throws IndexOutOfBoundsException if the index is larger than the length of the list minus 1.
 */
```

4. How can this specification be fixed? Circle one. (4pts)

   A. Add a precondition: the index must be less than the size of the list.
   B. Add a precondition: the index must be greater than or equal to 0.
   C. Add a param clause.
   D. Add a precondition: the ImMuList must be nonempty.
   E. The specification is fine; nothing needs to be added.
/** Immutable mutable list. */
public interface ImMuList<E> {
    public boolean isEmpty();
    public ImMuList<E> prepend(E value);
    public ImMuList<E> append(E value);
    public E get(int i);
}

/** Represents the empty list. */
public class ImMuEmpty<E> implements ImMuList<E> {
    public boolean isEmpty() {
        return true;
    }
    public ImMuList<E> prepend(E value) {
        return new ImMuCons<E>(value, this);
    }
    public ImMuList<E> append(E value) {
        return new ImMuCons<E>(value, this);
    }
    public E get(int i) {
        throw new IndexOutOfBoundsException();
    }
}

/** Represents a non-empty list. */
public class ImMuCons<E> implements ImMuList<E> {
    public E first;
    public ImMuList<E> rest;
    public ImMuCons(E first, ImMuList<E> rest) {
        this.first = first;
        this.rest = rest;
    }
    public boolean isEmpty() {
        return false;
    }
    public ImMuList<E> prepend(E value) {
        return new ImMuCons<E>(value, this);
    }
    public ImMuList<E> append(E value) {
        if (rest.isEmpty()) {
            rest = new ImMuCons<E>(value, rest);
        } else {
            rest.append(value);
        }
        return this;
    }
    public E get(int i) {
        if (i == 0) {
            return first;
        } else {
            return rest.get(i-1);
        }
    }
}
For each of the methods below, fill in the blanks with the list returned by the method. If there are more blanks than items in the list, leave them empty. For example, if a method returned:

```
return new ImMuCons(1, new ImMuCons(2, new ImMuEmpty()));
```

You would write: 1, 2, _, _, _

5. **public static** ImMuList<String> listTest1() {
   ImMuList<String> list = new ImMuCons<String>("b", new ImMuEmpty<String>());
   list.append("c");
   list.prepend("a");
   return list;
}

6. **public static** ImMuList<String> listTest2() {
   ImMuList<String> list = new ImMuCons<String>("b", new ImMuEmpty<String>());
   return list.append("c").prepend("a");
}

7. **public static** ImMuList<String> listTest3() {
   ImMuList<String> list = new ImMuEmpty<String>();
   list.append("c");
   list.prepend("b");
   return list.prepend("a");
}
State Machines & Git

Consider a Git repository with one file A.txt, which is the only file in the repository. Assume A.txt was previously added and committed with its current contents. Suppose we can perform only the following three actions:

**add** is defined as: `git add A.txt`

**commit** is defined as: `git commit -m "Working"`

**append** is defined as: `append-line-to A.txt`

Recall that `git add` stages changes for a future commit, and `git commit` commits changes that were previously staged (the `-m` option supplies a commit message without opening an editor). And suppose `append-line-to` is a command with the following specification:

**append-line-to filename** appends a single non-empty line of new text to the end of the file given by `filename`.

We will model this repository as a state machine with the following four states:

- **clean**: working directory has no changes
- **unstaged**: has unstaged changes only
- **both**: has both unstaged changes and staged changes
- **staged**: has staged changes only

1. Construct the state machine model by filling in the table below (page 9 is blank for scratch space). (12pts)

   For each state and each action, write either:
   
   **NONE** to indicate there is no outgoing edge from that state with that action
   
   or, the name(s) of the state(s) reachable from that state with that action

<table>
<thead>
<tr>
<th>from state</th>
<th>action</th>
<th>to state(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>clean</td>
<td>add</td>
<td>clean</td>
</tr>
<tr>
<td>clean</td>
<td>commit</td>
<td>clean</td>
</tr>
<tr>
<td>clean</td>
<td>append</td>
<td>clean</td>
</tr>
<tr>
<td>unstaged</td>
<td>add</td>
<td>unstaged</td>
</tr>
<tr>
<td>unstaged</td>
<td>commit</td>
<td>unstaged</td>
</tr>
<tr>
<td>unstaged</td>
<td>append</td>
<td>unstaged</td>
</tr>
<tr>
<td>both</td>
<td>add</td>
<td>both</td>
</tr>
<tr>
<td>both</td>
<td>commit</td>
<td>both</td>
</tr>
<tr>
<td>both</td>
<td>append</td>
<td>both</td>
</tr>
<tr>
<td>staged</td>
<td>add</td>
<td>staged</td>
</tr>
<tr>
<td>staged</td>
<td>commit</td>
<td>staged</td>
</tr>
<tr>
<td>staged</td>
<td>append</td>
<td>staged</td>
</tr>
</tbody>
</table>
Equality

Consider the two datatypes below, designed to represent alphanumeric strings (strings consisting of numbers and letters). Assume the comments correctly document the AF and RI for each datatype.

```java
class MyString1 {
    // AF: the string is the concatenation of only the alphanumeric characters in "chars".
    private final char[] chars;
    // ... methods ...
}
class MyString2 {
    // AF: the string is the concatenation of "first" with "rest",
    // or the string with only "first" if "rest" is null.
    // RI: "first" is alphanumeric.
    private final char first;
    private final MyString2 rest;
    // ... methods ...
}
```

1. Which of the following is true? Circle one: (2pts)
   A. MyString1 can represent more abstract values than MyString2
   B. MyString2 can represent more abstract values than MyString1
   C. MyString1 and MyString2 can represent the same set of abstract values

For each of the following questions, circle True or False.

2. MyString1 has multiple representation values with the same abstract value (2pts)
   A. True  B. False

3. MyString2 has multiple representation values with the same abstract value (2pts)
   A. True  B. False

4. MyString1 has multiple abstract values with the same representation value (2pts)
   A. True  B. False

5. MyString2 has multiple abstract values with the same representation value (2pts)
   A. True  B. False

6. Implement the equals method for MyString2 by circling all of the lines of code below that should be included: (4pts)
   ```java
   public boolean equals(Object obj) {
       A. if (this == null) { return false; }
       B. if (obj == null) { return false; }
       C. if ( !(MyString2)obj) { return false; }
       D. if ( !(obj instanceof MyString2)) { return false; }
       E. MyString2 other = (MyString2)obj;
       F. return this.first == other.first && this.rest == other.rest;
       G. return this.first == other.first && this.rest.equals(other.rest);
       H. return this.first.equals(other.first) && this.rest.equals(other.rest);
   }
   ```
This blank page is for your work or diagrams; what you write here will *not* be graded.