Quiz 2 (December 4, 2017)

Your name:____

Your Kerberos username:________________________________

You have 50 minutes to complete this quiz. It contains 10 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: Recursive data types	15
2: Recursive data types	23
3: Thread safety	25
4: Map-filter-reduce	22
5: Grammars	15
Total	100

This quiz uses the same abstract data type as Quiz 1: a **shopping list** is an **ordered** list of **unique** items together with a **positive integer** quantity for each listed item (we will not consider units for quantities, *e.g.* 1 oz. vs. 1 lb. vs. 1 box, just a number).

For example, someone who plans to buy a gallon of milk, a small bunch of bananas, and a box of Klondike bars, might have this shopping list:

- `1% milk (1)`
- `Banana (3)`
- `Klondike bars (1)`

Shopping lists are ordered, so the following is a **different** shopping list. It has the same items and quantities, but in a different order:

- `Banana (3)`
- `Klondike bars (1)`
- `1% milk (1)`

This is **not** a shopping list:

- `Banana (1)`
- `Banana (1)`
- `Banana (1)`
- `Klondike bars (1)`
- `1% milk (1)`

And neither is this:

- `1% milk (0)`

An unlisted item has zero quantity, so the quantity of `Oranges` on every list on this page is 0.

Problem 1 (Recursive data types) (**15 points**).

The problems of this exam refer to the code for `ShoppingList` on page 9, which you may detach.

(a) Write the data type definition for `ShoppingList`.

(b) Here is a new operation proposed for the `ShoppingList` type.

`atMost`: `ShoppingList list` × `String anItem` × `int count` → `ShoppingList`

requires true
effects returns a shopping list identical to `list`, except that `anItem`'s quantity on the returned list is the smaller of `count` and its quantity on `list`, i.e.
`returnedList.howMany(anItem) == min(count, list.howMany(anItem))`

Change just the **precondition** of this specification, as little as possible, so that the postcondition can always be satisfied:

Circle whether your new **specification** is STRONGER or WEAKER than the original **specification**, and explain why in one sentence:

spec is STRONGER / WEAKER because:

Problem 2 (Recursive data types) (**23 points**).
Using your revised spec for `atMost`, fill in the blanks below to implement the new operation as an instance method of `ShoppingList`. Assume the other code in those classes remains unchanged.

```
public interface ShoppingList {
    ...
    /** assume specification comment is here, don't write it */

}

class Empty implements ShoppingList {
    ...
    @Override public /* assume method signature is here, don't write it */ {

    }
}

class AnotherItem implements ShoppingList {
    ...
    @Override public /* assume method signature is here, don't write it */ {
        if (anItem.equals(this.item)) {

        } else {

        }
    }
}
```

Problem 3 (Thread safety) (**25 points**).
This problem refers to the code for `goShopping()` on page 10, which you may detach.

For each of these comments about the thread safety of the code below, circle AGREE or DISAGREE and explain why in one sentence.

(a) The `i` variable is threadsafe because it is confined.

AGREE / DISAGREE because:

(b) The `itemLocations` map is threadsafe because `goShopping` doesn't call any of its mutators.

AGREE / DISAGREE because:

(c) The `shoppers` list is threadsafe because it uses threadsafe data types.

AGREE / DISAGREE because:

(d) The code shown has a race condition.

AGREE / DISAGREE because:

(e) Declaring `goShopping()` with `public synchronized` would make `goShopping()` threadsafe.

AGREE / DISAGREE because:

Problem 4 (Map-filter-reduce) (**22 points**).

Let's define `map` and `filter` operations for `ShoppingList` that examine and operate on the items (but not their quantities). For example, `map` could be used to replace `Milk` with `1% Milk`, and `filter` could be used to filter to only the vegetarian items on the shopping list.

(a) Write mathematical type signatures (not Java method declarations) for the `map` and `filter` operations of `ShoppingList`. One argument of each operation should be a function, and all types should be specific to the shopping list problem, not generic type variables like <E> or <T>.

`map`: $\boxed{} \times \boxed{} \rightarrow \boxed{}$

`filter`: $\boxed{} \times \boxed{} \rightarrow \boxed{}$

(b) Assuming that `map` and `filter` are now implemented as instance methods of `ShoppingList`, write Java code to convert the list `input`:

- `small apples (5)`
- `green apples (3)`
- `leafy lettuce (1)`
- `yucky meat (3)`
- `yummy meat (7)`

into the list `output`:

- `apples (8)`
- `lettuce (1)`
- `meat (7)`

Use the fewest Java expressions you can.

```
ShoppingList input = ...; // abstract value shown above

ShoppingList output =
```

Problem 5 (Grammars) (**15 points**).

(a) Complete this partial grammar so that it can recognize shopping lists. It should match as many valid shopping lists as possible, and exclude as many invalid lists as possible. For example, the grammar **should match**:

- `Milk (1)`
- `Banana (30)`
- `Klondike bars (1)`

but **not match**:

- `Milk (0)`

Here is the partial grammar, with root nonterminal `list`. Complete the grammar by adding new rules; don't change existing rules.

```
list ::= ('• ' item ' ' quantity '\n')*;
item ::= [A-Za-z ]*;
```

(b) Read the grammar carefully, and then write Java code below to construct a `ShoppingList` that this grammar **cannot** match. (If you can't think of one, say that, but also write Java code that constructs some `ShoppingList`.)

```
ShoppingList list =
SFB
ETU
RFC
```

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

```java
/** Immutable shopping list. */
public interface ShoppingList {

    /** @return an empty shopping list */
    public static ShoppingList empty() {
        return new Empty();
    }

    /** @return this list with 1 instance of item added */
    public ShoppingList another(String item);

    /** @return the items on this list */
    public Set<String> items();

    /** @return the quantity of item on this list, >= 0 */
    public int howMany(String item);

    //...
}

class Empty implements ShoppingList {
    public Empty() {
    }

    // ...
}

class AnotherItem implements ShoppingList {
    private final String item;
    private final ShoppingList list;

    public AnotherItem(String item, ShoppingList list) {
        this.item = item;
        this.list = list;
    }

    // ...
}
```

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

```java
/** Aisle is a threadsafe mutable data type representing an aisle in a store */
public class Aisle { ... }

/** ShoppingCart is a threadsafe mutable data type representing a shopping cart */
public class ShoppingCart { ... }


public class Store {

    private final Map<String,Aisle> itemLocations;
    ... // other fields and operations

    /**
     * Requires that this store has sufficient quantity of every item on list.
     * Modifies the cart and this store to move the quantity of each item on list
     * from the store to the cart.
     */
    public void goShopping(final ShoppingList list,
                           final ShoppingCart cart) {
        final List<Thread> shoppers = new ArrayList<>();
        for (final String item: list.items()) {
            Thread shopper = new Thread(() -> {
                Aisle aisle = itemLocations.get(item);
                for (int i = 0; i < list.howMany(item); ++i) {
                    if (aisle.howMany(item) > 0) {
                        cart.add(item);
                        aisle.remove(item);
                    }
                }
            });
            shopper.start();
            shoppers.add(shopper);
        }
        for (Thread shopper : shoppers) { shopper.join(); }
    }
}
```