Solutions to Quiz 2 (December 4, 2017)

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 6, which you may detach.

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

Solution. `ShoppingList = Empty() + AnotherItem(item:String, list: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:

Solution. **requires** `count` ≥ 0 ■

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

Solution. spec is WEAKER because: (any of these answers is good)

- precondition is stronger
- the client has more restrictions
- any implementation that satisfies the original spec also satisfies the new spec
- the new spec allows at least one implementation, while the original spec's legal implementations were the empty set, so the implementations of the new spec is a superset of the impementations of the original spec

■

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 */
```

Solution. `public ShoppingList atMost(String anItem, int count);` ■

```
}

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

Solution.
`return this;`
or
`return new Empty();`
or
`return ShoppingList.empty();`
or
`return empty();`

■

```
    }
}

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

Solution. This is approach A:

```
            if (count == 0) return this.list.atMost(anItem, 0);
            else return new AnotherItem(this.item, this.list.atMost(anItem, count-1));
```

This is approach B:

```
            if (this.howMany(anItem) <= count) return this;
            else return this.list.atMost(anItem, count);
```

Approach A removes the final occurrence(s) of `anItem` on the list, while approach B removes the initial occurrence(s). Removing initial or final occurrences may change the item's position on the abstract *ordered* list, which the spec for `atMost` did not allow. But that requires knowing the abstraction function for `ShoppingList`, which was omitted from the provided code, so both approaches were accepted for full credit.

An iterative solution using `items()` does not receive full credit because the unordered `Set` it returns clearly discards the ordering of the `ShoppingList`, so the order of the resulting list can never be guaranteed to match the order of the original list, contrary to the spec for `atMost`.

■

```
        } else {
```

Solution.

```
            return new AnotherItem(this.item, this.list.atMost(anItem, count));
```

or

```
            return this.list.atMost(anItem, count).another(this.item);
```

■

```
        }
    }
}
```

Problem 3 (Thread safety) (**25 points**)**.**
This problem refers to the code for `goShopping()` on page 7, 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.

Solution. AGREE, because `i` is a local variable whose scope is the **`for`** loop, so no other thread has access to it. ■

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

Solution. DISAGREE, because: (any of these answers is good)

- the map could have beneficent mutation that happens when an observer is called
- other threads calling other operations of ‘Store‘ could call mutators of the map

■

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

Solution. DISAGREE, because `ArrayList` is *not* a threadsafe datatype. It happens that `shoppers` is only accessed by the original thread that called `goShopping()`, but it is still in scope for the new threads, so it is not actually confined, and the potential for a bug exists. ■

(d) The code shown has a race condition.

Solution. AGREE, because multiple threads may see `aisle.howMany(item) > 0` but then aren't all able to `aisle.remove(item)`.

(The threads started by `goShopping()` will not race in this way, because each thread is shopping for a different item, but multiple clients calling `goShopping()` can.)

■

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

Solution. DISAGREE. The **`synchronized`** keyword will solve the race condition between multiple clients calling `goShopping()`, but it will not prevent the threads created internally by `goShopping()` from using its rep unsafely. ■

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`: $\times \rightarrow$

`filter`: $\times \rightarrow$

Solution.

One possible answer:

`map`: `ShoppingList` $\times$ (`String` $\rightarrow$ `String`) $\rightarrow$ `ShoppingList`

`filter`: `ShoppingList` $\times$ (`String` $\rightarrow$ **`boolean`**) $\rightarrow$ `ShoppingList`

Another answer using Java's functional interfaces:

`map`: `ShoppingList` $\times$ `Function<String,String>` $\rightarrow$ `ShoppingList`

`filter`: `ShoppingList` $\times$ `Predicate<String>` $\rightarrow$ `ShoppingList`

■

(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 =
```

Solution.

```
input.filter( item -> !item.equals("yucky meat") )
     .map( item -> item.substring(6) );
```

Syntactic variations on the lambda expression are possible, as are function bodies with equivalent behavior on these particular items, e.g. `!item.contains("yucky")` or `item.split(" ")[1]`. ■

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 ]*;
```

Solution. `quantity ::= '(' [1-9] [0-9]* ')';` ■

Solution. ■

(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 =
```

Solution. Any answer with non-letter/non-space characters in an item name is unparseable by the grammar, e.g.:

```
empty().another("1% milk");
```

■

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

```
1  /** Immutable shopping list. */
2  public interface ShoppingList {

3      /** @return an empty shopping list */
4      public static ShoppingList empty() {
5          return new Empty();
6      }

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

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

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

13     //...
14 }

15 class Empty implements ShoppingList {
16     public Empty() {
17     }

18     // ...
19 }

20 class AnotherItem implements ShoppingList {
21     private final String item;
22     private final ShoppingList list;

23     public AnotherItem(String item, ShoppingList list) {
24         this.item = item;
25         this.list = list;
26     }

27     // ...
28 }
```

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

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

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


5  public class Store {

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

8      /**
9       * Requires that this store has sufficient quantity of every item on list.
10      * Modifies the cart and this store to move the quantity of each item on list
11      * from the store to the cart.
12      */
13     public void goShopping(final ShoppingList list,
14                            final ShoppingCart cart) {
15         final List<Thread> shoppers = new ArrayList<>();
16         for (final String item: list.items()) {
17             Thread shopper = new Thread(() -> {
18                 Aisle aisle = itemLocations.get(item);
19                 for (int i = 0; i < list.howMany(item); ++i) {
20                     if (aisle.howMany(item) > 0) {
21                         cart.add(item);
22                         aisle.remove(item);
23                     }
24                 }
25             });
26             shopper.start();
27             shoppers.add(shopper);
28         }
29         for (Thread shopper : shoppers) { shopper.join(); }
30     }
31 }
```