CSCI 3155: Principles of Programming Languages
Exercise sheet #14
28 June 2007

Group name:_____

Abstract Datatypes

This exercise is partially a lab exercise. You will have to use both SML and an object-oriented language (preferrably C++ or Java) for the practical part of this exercise. Leave the resulting source code in your home directory, underneath a directory `worksheet-14`, or send it to the instructor per e-mail (creichen@machine.cs.colorado.edu).

Exercise 1. Modern object-oriented languages provide the visibility annotations public, protected, and private for fields, methods, and constructors.

(a) (**Skills 15.1, 15.2**) Explain the utility of public, protected, and private methods. For each, explain why they are useful and how they relate to Abstract Data Types. Give examples where opportune.

(b) Examine *constructors*. Which of the three kinds of visibility are useful for constructors, and why?

Exercise 2. Early versions of the Java programming language defined their container classes with interfaces such as the following:

```
public class Box
{
    private Object o = null;
    public void put(Object new_o) { this.o = new_o; }
    public Object get() { return o; }
}
```

Later versions used different container classes, by utilising *Parametric Polymorphism*, provided in the form of *Generics*.

(a) (**Skill 15.3**) "Generify" the above class (on paper): change it to use parametric polymorphism.

(b) Describe (on paper) a client of your generic Box class.

(c) (**Skill 15.1**) Why are generics useful in this case? Illustrate using the client you sketched.

Exercise 3. Sometimes systems receive processing requests faster than they can process them. For these kinds of situations, we use two datatypes: *Queue*s and *Priority Queue*s.

The *Queue* is a data type that allows us to insert elements into the queue, and to remove these elements again. For a Queue, the order in which the elements are removed is the same as the order in which they were inserted.

There are several well-known approaches for implementing queues. Four of the easiest approaches are listed below:

- Doubly-Linked Lists: Store the data in a doubly-linked list. Attach new data to the first node, and remove it from the last node of the doubly-linked list (O(1) for both operations)
- One singly-linked list: append each element to the end of the list (O(n) insert, O(1) remove).
- Two singly-linked lists: Use two lists, in and out.

  Whenever an element is inserted, prepend it to the in list.

  Whenever an element is removed, check the out list. If the out list is not empty, remove and return the first element of the out list.

  If you try to remove an element and the out list is empty, reverse the in list and set the out list to this reversed in list. Then empty the in list. Check again if the out list is now empty; if it is, return failure. Otherwise remove and return the head of the out list. (O(1) insert, O(n) remove, but summed up over m runs it is O(m) for both operations.)
- Use a built-in datatype that already provides the neccessary functionality.

(a) (**Skill 15.2, Skillset 13**) Implement a queue as an Abstract Data Type, in SML. Note that SML has no built-in null values to indicate failure (in the case of no elements being present). You may instead use the predefined option type:

```
datatype 'a option = NONE
                   | SOME of 'a
```

(b) (**SKill 15.2**) Implement a queue as an Abstract Data Type, in Java or C++.

Exercise 4. The *Priority Queue* is a modification of the Queue: In a Priority Queue, elements are removed based on their *priority*. Each element has a priority associated with it that indicates whether we should remove it earlier or later. A Priority Queue, then, has the following operations:

- newQueue, which constructs a new priority queue
- insert(q, a), which inserts an element a into the priority queue q,
- remove(q), which removes the *highest-priority* element in the queue.

In practice, we usually don't give numeric priorities to elements; instead, we expect them to provide a *comparison mechanism*. In C++, we might expect the type of elements to provide a compare_to member fnuction or to overload operator::<. In Java, we would expect the type of elements to implement the interface java.util.Comparator. In SML, we would expect the user to provide us with an explicit comparison function.

(a) (**Skill 15.2, Skillset 13**) Implement an SML signature for Priority Queues. Make sure that the SML system finds no errors in the interface definition. Explain how and where the comparison mechanism becomes manifest in your interface.

You need not implement a corresponding structure.

(b) (**Skills 15.2, 15.2, 17.4**) Implement an abstract Java or C++ class for Priority Queues. The class should look like a real attempt to implement the queues, except for lacking the actual implementation. Explain how and where the comparison mechanism becomes manifest.

(c) In practice, there are four different times at which we can bind the comparison mechanism to the Priority Queue:

- Whenever we construct an instance of the Priority Queue (passing the comparison function as a parameter to the constructor)
- Whenever we construct a type that we wanted to use in the Priority Queue (having the type implement a Comparator)
- Whenever we implement the Priority Queue (only for monomorphic Priority Queues)
- Whenever we instantiate the type constructor of the Priority Queue (as when using operator::<)[1]

Compilers can (often considerably) speed up such generic code if they know statically which comparison mechanism we use.

Consider these four approaches. Which approaches fix the comparison mechanism at compile time, which approaches allow the comparison mechanism to be decided at runtime?

(d) Consider again the four possible binding times for the comparison mechanism. For each, list at least one disadvantage.

[1] This approach is also used with C++ templates and ML functors.