Candidate number: 2031B
Paper 1
Question 3

a.

i. Call the constructor of the parent class

ii. The current instance of Element being constructed

iii. It is a sanity check. It makes it so that if no method is being overridden, the compiler throws an error, so you know there is a mistake. It does not actually affect the execution of working code.

iv.

```
class Element {
    final int item;
    final Element next;

    Element(int item, Element next) {
        super();
        this.item = item;
        this.next = next;
    }

    @Override
    public String toString() {
        return item + " " + (next == null ? "" : next);
    }
}
```

b.

```java
class FuncList {
    private Element myHead;

    public FuncList() {

    }

    private FuncList(Element head) {
        this.myHead = head;
    }

    public int head() {
        if (myHead == null) {
            throw new RuntimeException("The list is empty");
        } else {
            return myHead.item;
        }
    }

    public FuncList tail() {
        if (myHead == null) {
            throw new RuntimeException("The list is empty");
        } else {
            return new FuncList(myHead.next);
        }
    }

    public FuncList cons(int x) {
        Element newHead = new Element(x, this.myHead);
        return new FuncList(newHead);
    }

}
```

c.

i. This is because whichever class is used for T might not be immutable. The programmer could hold a reference to one of the instances of T held in the list and modify it from there, thus modifying the list contents. This could be remedied by requiring T to implement some interface with a method called copy which would create a copy of the object (with a different reference) and then the cons method could call copy on the object before passing it to the Element constructor.

ii. This is not necessary in this case. Suppose there exists a class B which inherits from A. If covariance of generic types was allowed, FuncList<B> would inherit from FuncList<A>.

Imagine an variable x of type FuncList<A>, whose reference actually points to an object of type FuncList<B>. Calling x.head() would return an object of type B, where an object of type A is expected. This is not an issue as B inherits from A

Calling x.tail() returns an object of type FuncList<B> where one of type FuncList<A> was expected. This is not an issue as we have assumed that covariance of generic types exists.

Calling x.cons with a parameter of type B simply returns another object of type FuncList<B> which is not a problem. Calling x.cons with a parameter of type A returns an object of type FuncList<A> whose tail is of type FuncList<B> (but since we have assumed that covariance of generic types is allowed, this is not a problem).

There is no method or property of the FuncList class which would create a problem if covariance of generic types was allowed, and so the restriction is not necessary in this case.