19 Design Pattern: Adapters

We have been playing with a particular design pattern, the Iterator design patter, for several weeks. The purpose of that pattern is to implement a certain pattern of behavior, namely, the behavior "provide sequential access to all the entries in an aggregate structure".

We focused on a particular class of iterators, functional iterators. Recall that our functional iterators use the following interface:

```
public interface FuncIterator<T> {
  public boolean hasElement ();
  public T element ();
  public FuncIterator<T> moveToNext ();
}
```

The fact that these iterators were functional is reflected in the interface by having an explicit method to make the iterator move to the next element. In particular, calling the `element()` method repeatedly always returns the same answer. Putting it differently, functional iterators are immutable. (The term *functional* often carries the meaning of *immutable* in programming.)

Our running example, as usual, will use the `List<A>` ADT equipped with functional iterators. For variety's sake, I use nested classes to implement functional iterators.

```
public abstract class List<A> {

  public static <B> List<B> empty () {
    return new EmptyList<B>();
  }
  public static <B> List<B> cons (B i, List<B> l) {
    return new ConsList<B>(i,l);
  }

  public abstract boolean isEmpty ();
  public abstract A first ();
  public abstract List<A> rest ();
  public abstract String toString ();
```

```
  public abstract FuncIterator<A> funcIterator ();
}


class EmptyList<A> extends List<A> {

  public EmptyList () {}

  public boolean isEmpty () {
    return true;
  }
  public A first () {
    throw new Error ("EmptyList.first()");
  }
  public List<A> rest () {
    throw new Error ("EmptyList.rest()");
  }
  public String toString() {
    return "";
  }
  public FuncIterator<A> funcIterator () {
    return new FIterator<A>();
  }

  private static class FIterator<A> implements FuncIterator<A> {

    public FIterator () {}

    public boolean hasElement () {
      return false;
    }
    public A element () {
      throw new java.util.NoSuchElementException
                                ("EmptyList.FIterator.element()");
    }
    public FuncIterator<A> moveToNext () {
      throw new java.util.NoSuchElementException
                                ("EmptyList.FIterator.moveToNext()");
    }
  }
}
```

```
class ConsList<A> extends List<A> {
  private A first;
  private List<A> rest;

  public ConsList (A f, List<A> r) {
    first = f;
    rest = r;
  }

  public boolean isEmpty () {
    return false;
  }
  public A first () {
    return first;
  }
  public List<A> rest () {
    return rest;
  }
  public String toString() {
    return first + "  " + rest;
  }
  public FuncIterator<A> funcIterator () {
    return new FIterator<A>(first,rest);
  }

  private static class FIterator<A> implements FuncIterator<A> {
    private A current;
    private List<A> rest;

    public FIterator (A c, List<A> r) {
      current = c;
      rest = r;
    }

    public boolean hasElement () {
      return true;
    }
    public A element () {
      return current;
    }
    public FuncIterator<A> moveToNext () {
```

```
    return rest.funcIterator();
   }
  }
}
```

We will also make use of the following function to print the elements from a functional iterator:

```
public static <A> void printElements (FuncIterator<A> fIter) {
  FuncIterator<A> it = fIter;
  while (it.hasElement()) {
    System.out.println(" Element = " + it.element());
    it = it.moveToNext();
  }
}
```

19.1 Java Iterators

Now, you are all aware that Java has a built-in notion of iterators, which in particular the aggregator classes in the Java Collections Framework all implement.

Java iterators are similar in spirit to the functional iterators, and therefore are a form of Iterator design pattern. The big difference is that they are *mutable*, which is why sometimes we'll call them *mutable iterators*. Here is the interface Java implements (in `java.util.Iterator`):

```
public interface Iterator<T> {
  public boolean hasNext ();
  public T next ();
  public void remove ();
}
```

Method `hasNext()` is similar to `hasElement()` in functional iterators, it returns whether there is an element left to return from the iteration. Method `next()` returns the next element in the iteration, like `element()`, except is also automatically moves the iterator to the next object. The final method `remove()`, can be used for removing elements during iteration, but we won't be bothering with it. It is optional anyways according to the Java documentation, and an implementation is free to throw the `UnsupportedOperationException` when `remove()` is called.

Calling `next()` does not produce a new instance of the iterator to point to the next element, unlike functional iterators: the current instance of the iterator is mutated so that it now points to the next element. This means, among other things, that invoking `next()` twice on the same iterator will generally yield two different values.

Why do we care about Java iterators at all, aside from the fact that we need to know about them to iterate over Java Collections classes? Putting it differently, if you already have functional iterators for a particular ADT you've designed, why should you care about Java iterators?

One advantage of Java iterators is that Java offers *syntactic support* for them. Consider how you would use a Java iterator, for instance to iterate over all the elements of a `LinkedList<Integer>`:

```
LinkedList<Integer> lst = ...  // some code to create a list

Iterator<Integer> it = lst.iterator(); // get an iterator on the list
while (it.hasNext()) {
  Integer nxt = it.next();
  // some code acting on each integer in the list, e.g.:
  System.out.println ("Entry = " + nxt.toString());
}
```

A class implements `Iterable<T>` (available as `java.lang.Iterable`) if it has a method `iterator()` that can return an iterator for the class. Class `LinkedList<T>`, for instance, implements `Iterable<T>`. When you have a class implementing `Iterable<T>`, we can use a special for loop, called a foreach loop, as follows:

```
LinkedList<Integer> lst = ... // some code to create a list

for (Integer item : lst)
  System.out.println ("Entry = " + item.toString());
```

This is exactly equivalent to the while loop above. In fact, you can think of the Java statement

```
for (TYPE ITEM_VAR : COLL_EXP)
  STATEMENT
```

as shorthand for the longer

```
Iterator<TYPE> it = COLL_EXP.iterator();
while (it.hasNext()) {
  TYPE ITEM_VAR = it.next();
  STATEMENT;
}
```

Exercise 1: *Take the* `List<T>` *ADT we've been looking at the whole semester, and add a method* `public Iterator<T> iterator ()` *that creates a Java iterator for the instance*

at hand, and also implement that iterator. Make `List<T>` *implement* `Iterable<T>`*, and use foreach loops to write a few examples of iterating over lists.*

Having two kinds of iterators around is a bit of a mess. For instance, you might care about using a library that implements a functional iterator, while your code has utility procedures that use Java iterators, or vice versa.

It is of course possible to rewrite code, or reimplement libraries to use the kind of iterators your code is expecting.

Another possibility is to write a little class that *adapts* the objects returned by the library to interact better with your application. This is a general enough occurrence that we often call this an *Adapter design pattern*, although it is so obvious that the term seems like overkill.

Before working on adapting iterators and functional iterators, let's study adaptation in a simpler setting, namely streams.

19.2 Example: Adapters for Streams and Functional Iterators

As we saw in Lecture 15, streams are a lazy data structure that can be used to represent infinite sequences of values. We didn't emphasize this when we talked about streams, but streams (which are sequences of values) and functional iterators (which supply values in sequence) are similar to each other.

We have a bunch of code that can work with streams. We should be able to take an ADT with a functional iterator and "convert" that functional iterator into a stream of values, and conversely take a stream of values and turn it into a functional iterator that supplies those values when asked, in order.

Let's do so by writing adapters. Let's first deal with the easy case, writing an adapter to turn a stream into a functional iterator.

```
public class FIteratorFromStreamAdapter<A> implements FuncIterator<A> {
   private Stream<A> stream;

   private FIteratorFromStreamAdapter (Stream<A> s) {
      stream = s;
   }

   public static <B> FIteratorFromStreamAdapter<B> create (Stream<B> s) {
      return new FIteratorFromStreamAdapter<B>(s);
   }

   public boolean hasElement () {
      return true;   // a stream always has elements
```

```
    }

    public A element () {
        return stream.head();
    }

    public FuncIterator<A> moveToNext () {
        return FIteratorFromStreamAdapter.create(stream.tail());
    }
}
```

Let's try it out on some of the streams code we had in Lecture 15:

```
Stream<Integer> s1 = Stream.intsFrom(1);
Stream<Integer> s = Stream.sum(s1,s1);
FuncIterator<Integer> sIter = FIteratorFromStreamAdapter.create(s);

printElements(sIter);
```

This constructs the infinite stream `s` of even numbers starting from 2, creates a functional iterator out of the stream, and prints the elements from the functional iterator using the function `printElements()` we saw above. The result is, predictably:

```
Element = 2
Element = 4
Element = 6
Element = 8
Element = 10
Element = 12
Element = 14
Element = 16
Element = 18
Element = 20
Element = 22
Element = 24
Element = 26
...
```

and so on. Because the stream is infinite, the resulting functional iterator always has a next element, therefore `printElements()` never terminates.

So we can write an adapter to transform a stream into a functional iterator. We can also go the other way around, converting a functional iterator into a stream. The one difficulty here

is that a functional iterator may only supply finitely many values, while a stream is required to supply infinitely many values. So we need a way to "pad" the values provided by the functional iterator if it runs empty. What the best approach is depends on the context of use. What we'll do here it to use the `Option` ADT: the stream will wrap every value from the functional iterator with `Option.some()`, and when the functional iterator is empty the stream will start supplying `Option.none()` values.

```
public class StreamFromFIteratorAdapter<A> extends Stream<Option<A>> {
   private FuncIterator<A> iter;

   private StreamFromFIteratorAdapter (FuncIterator<A> it) {
      iter = it;
   }

   public static <B> StreamFromFIteratorAdapter<B> create (FuncIterator<B> it) {
      return new StreamFromFIteratorAdapter<B>(it);
   }

   public Option<A> head () {
      if (iter.hasElement())
         return Option.some(iter.element());
      else
         return Option.none();
   }

   public Stream<Option<A>> tail () {
      if (iter.hasElement())
         return StreamFromFIteratorAdapter.create(iter.moveToNext());
      else
         return this;
   }
}
```

The types tell the whole story here — look at the type of the creator: it takes a `FuncIterator<B>`, and returns a (subclass of) `Stream<Option<B>>`, thereby converting a functional iterator supplying values of type `B` into a stream of `Option<B>` values.

Some examples, using a simple list of strings, and the following `printFirstN()` function from Lecture 15, which prints the first n values of a stream:

```
public static <A> void printFirstN (Stream<A> s, int n) {

    Stream<A> temp = s;
```

```
        for (int i = 0; i < n; i++) {
            System.out.print(temp.head() + " ");
            temp = temp.tail();
        }
        System.out.println();
    }
```

And here is the sample code:

```
  List<String> el = List.empty();
  List<String> l = List.cons("goodbye",List.cons("cruel",List.cons("world",el)));
  Stream<Option<String>> sL = StreamFromFIteratorAdapter.create(l.funcIterator());

  printFirstN(sL,20);
```

with output the first 20 elements of the stream obtained from the functional iterator:

```
some(goodbye) some(cruel) some(world) none none none none none none
none none none none none none none none none none none
```

19.3 An Adapter for Java Iterators

Let's now turn to the problem of adapting between iterators and functional iterators, which forces us to deal with mutation.

Consider the `List<A>` implementation at the beginning of this lecture. We could implement a `iterator()` method in `List<A>` that returns an `Iterator<A>` instance, as Exercise 1 above asks you to do, but this would require us to rewrite a lot of iterator-related code, and we already have functional iterators implemented for lists.

So let's write an adapter class that wraps around a functional iterator and gives back a Java iterator that iterates over the same values as the functional iterator.

```
import java.util.Iterator;

public class IteratorFromFIteratorAdapter<A> implements Iterator<A> {
   private FuncIterator<A> fIter;

   private IteratorFromFIteratorAdapter (FuncIterator<A> f) {
      fIter = f;
   }

   public static <B> IteratorFromFIteratorAdapter<B> create (FuncIterator<B> f) {
```

```
    return new IteratorFromFIteratorAdapter<B>(f);
  }

  public boolean hasNext () {
    return fIter.hasElement();
  }
  public A next () {
    if (!(fIter.hasElement()))
      throw new java.util.NoSuchElementException
                          ("IteratorFromFIteratorAdapter.next()");
    A current = fIter.element();
    fIter = fIter.moveToNext();
    return current;
  }
  public void remove () {
    throw new UnsupportedOperationException
                        ("IteratorFromFIteratorAdapter.remove()");
  }
}
```

First, note that the above adapter is not specific to functional iterators over lists — it works for *any* functional iterator, and turns any such functional iterator into a Java iterator. Second, note the implementation of `next()`, which needs to update the field holding the functional iterator so that on the next call to `next()` another object is extracted from the iteration. You should understand the above code, perhaps using the framework I described last lecture for modeling mutability. To use this adapter class, we only need to create an instance of it, passing in a functional iterator.

We can try out this code as follows. First, define a function to print the elements given by a Java iterator

```
public static <A> void printElements (Iterator<A> iter) {
  while (iter.hasNext()) {
    System.out.println(" Element = " + iter.next());
  }
}
```

(Note that we *overload* the `printElements()` function — there is one for functional iterators, and one for Java iterators — the right one is called depending on the *compile-time type* of the argument we are passing to it.)

Next, some sample code, construct a list and iterating over it using both its functional iterator, and its Java iterator:

```
  List<String> el = List.empty();
  List<String> l = List.cons("goodbye",List.cons("cruel",List.cons("world",el)));
  System.out.println("l = " + l);
  System.out.println("-------------------------------------------------");
  System.out.println("Loop using functional iterator:");
  FuncIterator<String> it = l.funcIterator();
  printElements(it);
  System.out.println("Testing immutability:");
  System.out.println(" it.element() = " + it.element());
  System.out.println(" it.element() = " + it.element());
  System.out.println("-------------------------------------------------");
  System.out.println("Loop using Java iterator:");
  Iterator<String> it2 = IteratorFromFIteratorAdapter.create(it);
  printElements(it2);
  System.out.println("Testing mutability:");
  Iterator<String> it3 = IteratorFromFIteratorAdapter.create(it);
  System.out.println(" it3.next() = " + it3.next());
  System.out.println(" it3.next() = " + it3.next());
```

Which yields:

```
l = goodbye cruel world
-------------------------------------------------
Loop using functional iterator:
 Element = goodbye
 Element = cruel
 Element = world
Testing immutability:
 it.element() = goodbye
 it.element() = goodbye
-------------------------------------------------
Loop using Java iterator:
 Element = goodbye
 Element = cruel
 Element = world
Testing mutability:
 it3.next() = goodbye
 it3.next() = cruel
```

The above works without modifying the `List<A>` implementation. We can also take advantage of the iterator syntax in Java by adding an `iterator()` method to the List ADT — its implementation can simply rely on the adapter we defined above — and making `List<A>` implement `Iterable`:

```
import java.util.Iterator;
import java.lang.Iterable;

public abstract class List<A> implements Iterable<A> {

   public static <B> List<B> empty () {
      return new EmptyList<B>();
   }
   public static <B> List<B> cons (B v, List<B> l) {
      return new ConsList<B>(v,l);
   }

   public abstract boolean isEmpty ();
   public abstract A first ();
   public abstract List<A> rest ();
   public abstract String toString ();
   public abstract FuncIterator<A> fucIterator ();

   public Iterator<A> iterator () {
      return IteratorFromFIteratorAdapter.create(funcIterator());
   }
}
```

Note that we are using inheritance to make the code for `iterator()` available in both concrete subclasses, which do not need to change.

```
  List<String> el = List.empty();
  List<String> l = List.cons("goodbye",List.cons("cruel",List.cons("world",el)));
  System.out.println("l = " + l);
  System.out.println("-------------------------------------------------");
  System.out.println("Loop using Java iterator syntax:");
  for (String s : l)
    System.out.println(" Element = " + s);
```

which is as short as we can imagine, and which yields:

```
l = goodbye cruel world
-------------------------------------------------
Loop using Java iterator syntax:
 Element = goodbye
 Element = cruel
 Element = world
```

19.4 An Adapter for Functional Iterators

The above shows that we can adapt a functional iterator into a Java iterator. How about the other way around? More precisely, suppose we have an application geared towards using functional iterators, and we want to use one of the Java Collections classes, which all implement Java iterators.

It turns out to be a bit more complicated to adapt Java iterators into functional iterators, and not very robust, because as we saw last time, mutability is contagious. Here is one stab at such an adapter:

```
import java.util.*;

public class FIteratorFromIteratorAdapter<A> implements FuncIterator<A> {
   private Option<A> element;
   private Iterator<A> iter;

   private FIteratorFromIteratorAdapter (Iterator<A> it) {
      if (it.hasNext())
         element = Option.some(it.next());
      else
         element = Option.none();
      iter = it;
   }

   public static <B> FIteratorFromIteratorAdapter<B> create (Iterator<B> it) {
      return new FIteratorFromIteratorAdapter<B>(it);
   }

   public boolean hasElement () {
      return !(element.isNone());
   }
   public A element () {
      if (element.isNone())
         throw new NoSuchElementException("FIteratorFromIteratorAdapter.
    element()");
      return element.valOf();
   }
   public FuncIterator<A> moveToNext() {
      return FIteratorFromIteratorAdapter.create(iter);
   }
}
```

This adapter uses the `Option` ADT, which we saw in a previous lecture. The `Option` ADT lets us avoid the use of `null`, which is always error prone.

The idea behind the above adapter is that we have to make sure that we never call `next()` more than once for every element of the underlying Java iterator, because whenever we call `next()`, the iterator mutates. We therefore call `next()` once when we construct the functional iterator, storing the value (if there is one — if there is none, we also record that fact, using `Option.none()`) away so that when we ask for `element()` we do not need to query the underlying iterator, we just return the value we stored away. When we move the iterator to the next element, we simply construct a new adapter for the underlying iterator, which by that point is already pointing to the next element to be returned anyways.

Again, stare at the above code, and convince yourself that it works, and try to understand that it works. When you do, then you have nailed how mutation works.

Let's test the above adapter by constructing a functional iterator from the Java iterators of our `List<A>` implementation (which themselves were constructed from functional iterators, so that's many layers of indirection here), continuing on with the example from last section:

```
  System.out.println("Loop using (converted) functional iterator:");
  Iterator<String> it4 = IteratorFromFIteratorAdapter.create(it);
  FuncIterator<String> it5 = FIteratorFromIteratorAdapter.create(it4);
  printElements(it5);
  System.out.println("Testing immutability:");
  System.out.println(" it5.element() = " + it5.element());
  System.out.println(" it5.element() = " + it5.element());
```

which yields:

```
Loop using (converted) functional iterator:
 Element = goodbye
 Element = cruel
 Element = world
Testing immutability:
 it5.element() = goodbye
 it5.element() = goodbye
```

I claimed above that `FIteratorFromIteratorAdapter` is not very robust. What I mean is that there are ways to disrupt the result of iterating using a functional iterator obtained by `FIteratorFromIteratorAdapter`. The underlying Java iterator is shared with the resulting functional iterator, meaning that we can mutate the underlying Java iterator, and the result of the mutation will be visible in the functional iterator. Study the following example, again continuing the sample code above:

```
  System.out.println("Messing with iteration:");
```

```
  Iterator<String> it6 = IteratorFromFIteratorAdapter.create(it);
  FuncIterator<String> it7 = FIteratorFromIteratorAdapter.create(it6);
  System.out.println(" it7.element() = " + it7.element());
  System.out.println("Advancing underlying iterator -- " + it6.next());
  System.out.println(" it7.moveToNext().element() = " + it7.moveToNext().element());
```

This yields:

```
Messing with iteration:
 it7.element() = goodbye
Advancing underlying iterator -- cruel
 it7.moveToNext().current() = world
```

By accessing the Java iterator `it6`, we made the `it7` functional iterator skip an element — it thinks the element after `goodbye` is `world`. That's bad.

Exercise 2 (hard): *Write a different* `FIteratorFromIteratorAdapter<A>` *class that does not have this problem.*

19.5 Iterators for Trees

Adapting is useful to bridge gaps between code that provides a particular interface, and client code that expects a slightly different interface. Functional iterators and Java iterators being just one example of such bridging that is sometimes necessary.

Another use for adapters is that it sometimes helps simplify coding. Consider binary trees, with signature

```
  CREATORS:   <A> Tree<A> empty ()
              <A> Tree<A> node (A, Tree<A>, Tree<A>)

  ACCESSORS:  boolean isEmpty ()
              A root ()
              Tree<A> left ()
              Tree<A> right ()
              String toString ()
```

and the obvious specification. Defining Java iterators for trees is difficult — try it. Pick an order in which to traverse the tree (there are a few), and define an `iterator()` method for the implementation of trees you obtain by applying our Specification design pattern to the above specification.

There is another way to get Java iterators for trees, though. It is still difficult, but much easier, to define a functional iterator for trees. Once we have functional iterators, it suffices

to use the `IteratorFromFIteratorAdapter` class above to get Java iterators. Here is one implementation of functional iterators for trees, that implements a preorder iteration of the tree.[1]

```
import java.util.*;
import java.lang.Iterable;

public abstract class Tree<A> implements Iterable<A> {
   public static <B> Tree<B> empty () {
      return new EmptyTree<B>();
   }
   public static <B> Tree<B> node (B v, Tree<B> l, Tree<B> r) {
      return new NodeTree<B>(v,l,r);
   }

   public abstract boolean isEmpty ();
   public abstract A root ();
   public abstract Tree<A> left ();
   public abstract Tree<A> right ();
   public abstract String toString ();
   public abstract FuncIterator<A> funcIterator ();

   public Iterator<A> iterator () {
      return IteratorFromFIteratorAdapter.create(funcIterator());
   }
}


class EmptyTree<A> extends Tree<A> {
   public EmptyTree () {}

   public boolean isEmpty () { return true; }

   public A root () { throw new RuntimeException("EmptyTree.root()"); }

   public Tree<A> left () { throw new RuntimeException("EmptyTree.left()"); }

   public Tree<A> right () { throw new RuntimeException("EmptyTree.right()"); }
```

[1] A preorder iteration says that we return the value of a node before we iterate over its left or right subtrees. That's in contrast with a postorder iteration, where we first iterate over the left and right subtrees, then we return the value of the node, or an inorder iteration, where we first iterate over the left subtree, return the value of the node, then iterate over the right subtree.

```
    public String toString () { return ""; }

    public FuncIterator<A> funcIterator () {
      return new FIterator<A>();
    }

    private static class FIterator<A> implements FuncIterator<A> {
      public FIterator () {}

      public boolean hasElement () {
        return false;
      }
      public A element () {
        throw new NoSuchElementException("EmptyTree.FIterator.element()");
      }
      public FuncIterator<A> moveToNext () {
        throw new NoSuchElementException("EmptyTree.FIterator.moveToNext()");
      }
    }
}


class NodeTree<A> extends Tree<A> {
    private A root;
    private Tree<A> left;
    private Tree<A> right;

    public NodeTree (A v, Tree<A> l, Tree<A> r) {
        root = v;
        left = l;
        right = r;
    }

    public boolean isEmpty () { return false; }

    public A root () { return root; }

    public Tree<A> left () { return left; }

    public Tree<A> right () { return right; }

    public String toString () {
```

```
    return "[" + left() + "] " + root() + " [" + right() + "]";
  }

  public FuncIterator<A> funcIterator() {
    List<FuncIterator<A>> e = List.empty();
    List<FuncIterator<A>> remainder =
      List.cons(right().funcIterator(),e);
    return new FIterator<A>(root(),left().funcIterator(),remainder);
  }

  private static class FIterator<A> implements FuncIterator<A> {
    private A element;
    private FuncIterator<A> iterator;
    private List<FuncIterator<A>> remainder;

    public FIterator (A v, FuncIterator<A> n, List<FuncIterator<A>> r) {
      element = v;
      iterator = n;
      remainder = r;
    }

    public boolean hasElement () { return true; }

    public A element () { return element; }

    public FuncIterator<A> moveToNext () {
      // element left in current iterator?
      if (iterator.hasElement())
        return new FIterator<A>(iterator.element(),iterator.moveToNext(),remainder);
      // no element in current iterator -- any iterators remaining?
      if (remainder.isEmpty())
        return IteratorFromFIteratorAdapter.create(Tree.empty().funcIterator());
      // find next nonempty iterator
      FuncIterator<A> nextIter = remainder.first();
      List<FuncIterator<A>> nextRemainder = remainder.rest();
      while (!(nextIter.hasElement())) {
        if (nextRemainder.isEmpty())
          return IteratorFromFIteratorAdapter.create(Tree.empty().funcIterator());
        nextIter = nextRemainder.first();
        nextRemainder = nextRemainder.rest();
      }
      return new FIterator<A>(nextIter.element(), nextIter.moveToNext(),
```

```
                                nextRemainder);
    }
  }
}
```

The difficulty in this code is the implementation of `NodeTree.FIterator`, where we iterate over a node. The idea is that we return the value at the node, and then when we advance, we need to return a functional iterator that can yield the rest of the elements in the tree, namely the elements in the left subtree, and the elements in the right subtree. We know how to return an iterator for the elements of the left subtree — we just get the iterator that corresponds to the left subtree by calling `funcIterator()` on the left subtree. But we need to remember somewhere that when the iterator for the left subtree is exhausted, we have to look at the elements in the right subtree. And we need to do so recursively as we progress down the tree. So we record in `NodeTree.FIterator` the current element in the tree we're at, an "immediate" iterator that can immediately give us new elements, and the remainder of the iterators where we accumulate the list of all iterators that still have elements to yield, essentially corresponding to all the right subtrees we have encountered but not visited yet. (That's a brainful. And I still claim that this is easier than defining a mutable iterator directly.)

Method `iterator()` in `Tree<A>` just uses our functional iterators and the adapter class `IteratorFromFIteratorAdapter` from above.

Here are the iterators in action:

```
  Tree<Integer> et = Tree.empty();
  Tree<Integer> t66 = Tree.node(66,et,et);
  Tree<Integer> t87 = Tree.node(87,et,et);
  Tree<Integer> t = Tree.node(99,t66,t87);
  System.out.println("t = " + t);
  System.out.println("-------------------------------------------------");
  System.out.println("Loop using functional iterator:");
  FuncIterator<Integer> itt = t.funcIterator();
  printElements(itt);
  System.out.println("-------------------------------------------------");
  System.out.println("Loop using Java iterator:");
  for (Integer i : t)
    System.out.println(" Element = " + i);
```

(where we use the `printElements()` function defined in §19.3), which yields:

```
t = [[] 66 []] 99 [[] 87 []]
-------------------------------------------------
```

```
Loop using functional iterator:
 Element = 99
 Element = 66
 Element = 87
-------------------------------------------------
Loop using Java iterator:
 Element = 99
 Element = 66
 Element = 87
```