Laziness

For this lecture, I want to return to something that came up during the last homework, the third homework where you had to implement pictures. It is a sufficiently common phenomenon that we should make it precise.

A Slight Detour

First, a slight detour to talk about functional iterators, again. It's amazing how much mileage we can get out of this simple concept.

We've implemented functional iterators for Stacks a few times, you've implemented picture iterators. We're pros. But man, what work. If we follow the recipe to the letter, we get a concrete subclass for every creator for our ADT, and each concrete subclass has a method `getFuncIterator` that gives us back an iterator specific for the kind of data embodied by the concrete subclass. (See Lecture 11 for the Stack example.) That's a lot of classes to write. And the iterators are not doing things very much different than the class itself.

This general approach of having a `getFuncIterator` works well in general, and is doubly interesting for actual Java iterators, as we'll see. But if we have functional iterators and especially if we have immutable classes we want to iterator over, we can sometimes do something a bit more clever.

Instead of having a method `getFuncIterator` that extracts an iterator for an instance of an ADT, we can have *an instance of the ADT itself act as an iterator*. Let's do this for stacks.

Here is the specification I have in mind:

```
public static Stack empty ();
public static Stack push (Stack, int);
public boolean isEmpty ();
public int top ();
public Stack pop ();
public String toString ();
public boolean hasElement ();
public Integer current ();
public FuncIterator<Integer> advance ();

empty().isEmpty() = true
```

```
push(s,i).isEmpty() = false
push(s,i).top() = i
push(s,i).pop() = s
empty().toString() = "<bottom of stack>"
push(s,i).toString() = i + " " + s.toString()

empty().hasElement() = false
push(s,i).hasElement() = true
push(s,i).current() = i
push(s,i).advance() = s
```

Note that stacks directly implement the `FuncIterator<Integer>` interface. Note also that the `advance` method is required to return a `FuncIterator<Integer>`. It is perfectly fine to return a stack here (see the specification for `push(s,i).advance()` because a stack will be a subclass of `FuncIterator<Integer>`, and subclassing says that it's fine to return a class that is more precise than a class you are expecting to return. (Can you reason out why?)

Here is the corresponding implementation:

```
// Stack ADT with iterators
public abstract class Stack implements FuncIterator<Integer> {

  public static Stack emptyStack () {
    return new EmptyStack ();
  }

  public static Stack push (Stack s, int i) {
    return new PushStack (s,i);
  }

  public abstract boolean isEmpty ();
  public abstract int top ();
  public abstract Stack pop ();
  public abstract String toString ();
  public abstract boolean hasElement ();
  public abstract Integer current ();
  public abstract FuncIterator<Integer> advance ();
}

// Concrete subclass for empty()
//
class EmptyStack extends Stack {
  public EmptyStack () { }
```

```
  public boolean isEmpty () { return true; }

  public int top () {
    throw new IllegalArgumentException ("in EmptyStack.top()");
  }

  public Stack pop () {
    throw new IllegalArgumentException ("in EmptyStack.pop");
  }

  public String toString () { return "<bottom of stack>"; }

  public boolean hasElement () { return false; }

  public Integer current () {
    throw new IllegalArgumentException ("in EmptyStack.current()");
  }

  public FuncIterator<Integer> advance () {
    throw new IllegalArgumentException ("in EmptyStack.advance()");
  }
}

// Concrete subclass for push()
//
class PushStack extends Stack {
  private int topVal;
  private Stack rest;

  public PushStack (Stack s, int v) {
    topVal = v;
    rest = s;
  }

  public boolean isEmpty () { return false; }

  public int top () { return this.topVal; }

  public Stack pop () { return this.rest; }

  public String toString () { return this.top() + " " + this.pop(); }
```

```
  public boolean hasElement () { return true; }

  public Integer current () { return new Integer (this.topVal); }

  public FuncIterator<Integer> advance () { return this.rest; }
}
```

Note that we need the `current` method to return an `Integer`, that is, a boxed integer (wrapped inside a class) because `FuncIterator<T>` requires `T` to be a class type. That's an annoyance we have to deal with.

Streams

Back to what I really wanted to talk about.

In the homework, you had to define pictures. The key operation pictures was to draw them. Creators were provided to let you build more complicated pictures from simpler ones.

One question I was asked quite a few times is when the lines are computed. Let's pick an example. A basic picture is a grid. The flip operation takes a picture and produces a new "flipped" picture. Now, when we flip a grid, do we go in an actual do the flipping of the lines in the grid? If we flip a more complex picture, do we go into the picture, and flip all the grids inside it? That turns out to be *very hard* to get right. If you tried it, you know what I mean. If not, then try to think how that would work. In fact, the problem is that if we flip a rotated picture, we need to flip all the lines that have been rotated in the underlying grids.

The alternative is to be lazy. We do not do *any work* when creating a flipped picture. We simply record the underlying picture to be flipped. The work gets done if it's need, when we are asked to draw the flipped picture. Happily enough, this is exactly what comes out of the recipe I gave you. Here is the concrete subclass corresponding to the `flip` creator that my implementation uses:

```
... in the abstract Picture class:
   public static Picture flip (Picture p) {
     return new PicFlip(p);
   }
...

class PicFlip extends Picture {
   private Picture pic;

   public PicFlip (Picture p) {
```

```
    pic = p;
  }

  public void draw (Point a, Point b, Point c) {
    this.pic.draw(a.add(b),b.scale(-1),c);
  }
}
```

Amazingly simple. No work to be done when we construct a flipped picture. The work is done when we draw, and even then, it's not an amazing amount of work, because of the way we can use bounding boxes.

This general approach, only doing the work when it's needed, and not at creator-invocation time, is called *lazy construction*, or just plain *laziness*. (In contrast, if we build something at creator-invocation time, we often call it *eager construction*.)

Laziness is very powerful. In particular, it lets us work with infinite data rather straightforwardly. The classical example of infinite data is streams, which you can think of infinite lists. Of course, you cannot simply represent an infinite list directly by listing all its elements. Instead, a stream can be thought of simply as a "promise" to deliver its elements, if you ask for them. If you ask for the first 10 elements of the stream, it will compute them, and then give them to you. If you ask for the 200th element of the stream, it will compute it and give it to you. Until you ask for it, though, it is not explicitly represented. Just like in the pictures case, we can also create new streams from old streams, and here again, we do not do any work at creation time, only when we ask for elements of the stream.

We implement the following interface for streams:

```
public static Stream Cons (int, Stream);
public static Stream Tail (Stream);
public static Stream IntsFrom (int, Stream);
public static Stream Sum (Stream, Stream);
public static Stream Filter (Predicate<Integer>, Stream);

public boolean hasElement ();
public Integer current ();
public FuncIterator<Integer> advance();
// other operations?
// public boolean equals (Object);
// public String toString ();
```

Note that we have streams be their own iterators, following our approach above.

```
public abstract class Stream implements FuncIterator<Integer> {
```

```
  public static Stream cons (int i, Stream s) {
    return new StreamCons(i,s);
  }
  public static Stream tail (Stream s) {
    return new StreamTail(s);
  }
  public static Stream intsFrom (int i, Stream s) {
    return new StreamIntsFrom(i,s);
  }
  public static Stream sum (Stream s1, Stream s2) {
    return new StreamSum(s1,s2);
  }
  public static Stream filter (Predicate<Integer> p, Stream s) {
    return new StreamFilter(p,s);
  }

  public abstract boolean hasElement ();
  public abstract Integer current ();
  public abstract FuncIterator<Integer> advance();
}
```

We now implement the concrete subclasses. All are rather trivial, and all do not do any work until asked to what the `current()` element of a stream is, or what the `advance()` stream is.

```
// Concrete subclass for cons()
//
class StreamCons extends Stream {
  private Integer first;
  private Stream rest;

  public StreamCons (Integer f, Stream r) {
    first = f;
    rest = r;
  }

  public boolean hasElement () { return true; }

  public Integer current () { return this.first; }

  public FuncIterator<Integer> advance () { return this.rest; }
}
```

```
// Concrete subclass for tail()
class StreamTail extends Stream {
  private Stream str;

  public StreamTail (Stream s) {
    this.str = s;
  }

  public boolean hasElement () { return true; }

  public Integer current () { return this.str.advance().current(); }

  public FuncIterator<Integer> advance () {
    return Stream.tail((Stream) this.str.advance());
  }
}
```

For `StreamTail`, notice the cast required for `advance` to type check. You should be able to figure out why it's needed. (Hint: again, it's all about loss of typing information due to subclassing. See Lecture 12.)

```
// Concrete subclass for intsFrom()
//
class StreamIntsFrom extends Stream {
  private Integer n;

  public StreamIntsFrom (Integer n) {
    this.n = n;
  }

  public boolean hasElement () { return true; }

  public Integer current () { return this.n; }

  public FuncIterator<Integer> advance () {
    return Stream.intsFrom(this.n+1);
  }
}
```

```
// Concrete subclass for sum()
```

```
//
class StreamSum extends Stream {
  private Stream stream1;
  private Stream stream2;

  public StreamSum (Stream s1, Stream s2) {
    stream1 = s1;
    stream2 = s2;
  }

  public boolean hasElement () { return true; }

  public Integer current () {
    return stream1.current() + stream2.current();
  }

  public FuncIterator<Integer> advance () {
    return Stream.sum((Stream) stream1.advance(),
                      (Stream) stream2.advance());
  }
}
```

The `filter` creator is a bit more interesting. It takes a predicate as argument, and intuitively returns the stream of all elements of the supplied stream argument for which the predicate is true. In Scheme, we can represent a predicate as a function (or a lambda), In Java and most OO language, we need to wrap a function you want to pass as an argument in an object.

Let's define an interface for predicates over values of type `T`:

```
public interface Predicate<T> {
  public boolean pred (T arg);
}
```

Here is an example of a predicate over integers, that returns true if and only if an integer is not divisible by a given number:

```
public class NotDivByPredicate implements Predicate<Integer> {
  private Integer divisibleBy;

  private NotDivByPredicate (Integer y) {
    divisibleBy = y;
  }
```

```
  public static NotDivByPredicate mk (Integer y) {
    return new NotDivByPredicate(y);
  }

  public boolean pred (Integer x) {
    return (x % this.divisibleBy != 0);
  }
}
```

Thus, `NotDivByPredicate.mk(4)` returns a predicate that can used to test if an integer is not divisible by 4.

```
// Concrete subclass for filter()
//
class StreamFilter extends Stream {
  private Predicate<Integer> pred;
  private Stream str;

  private Filter (Predicate<Integer> p, Stream str) {
    this.pred = p;
    this.str = str;
  }

  public boolean hasElement () { return true; }

  public Integer current () {
    FuncIterator<Integer> tmp = this.str;
    // loop until we have found an element that satisfies the predicate
    while (!(this.pred.pred(tmp.current()))) {
      tmp = tmp.advance();
    }
    return tmp.current();
  }

  public FuncIterator<Integer> advance() {
    FuncIterator<Integer> tmp = this.str;
    // loop until we have found an element that satisfies the predicate
    while (!(this.pred.pred(tmp.current())))
      tmp = tmp.advance();
    return Stream.filter(this.pred, (Stream) tmp.advance());
  }
```

```
}
```

If we define a function for printing the first n elements out of a functional iterator over the integers:

```
public static void printFirstNFromIterator (FuncIterator<Integer> it, int
 n) {
  int i;
  FuncIterator<Integer> tmp = it;
  for (i=0; i<n; i++) {
    System.out.println("  " + tmp.current());
    tmp = tmp.advance();
  }
}
```

We can now try out something such as:

```
Stream s1 = Stream.intsFrom(3);
Stream s2 = Stream.intsFrom(10);

printFirstNFromIterator(Stream.sum(s1,s2),10);
```

This prints the first 10 elements of the stream obtained by summing the stream starting from 3 and the stream starting from 10. This yields:

```
13
15
17
19
21
23
25
27
29
31
```

As expected. Make sure you understand what is happening.

As a cute exampe, we can try to compute, using (a variant of) the Sieve of Eratosthenes, the stream of all prime numbers. The sieve computes the list of prime numbers by essentially starting with all integers from 2 on, and then keeping 2 and removing all multiples of 2, then moving to the next unremoved integer (3), keeping it and removing all multiples of 3,

moving to the next unremoved integer (5), keeping it and removing all multiples of 5, and so on. You can convince yourself that what you are left with is the stream of all prime numbres.

Here is a concrete subclass of `Stream` that implements the sieve—the creator, instead of being shoved away into `Stream`, is here available under the name `Sieve.mk()`.

```
// Concrete subclass for sieve (no creator in Stream class)
//
public class Sieve extends Stream {
  private Stream str;

  private Sieve (Stream s) {
    this.str = s;
  }

  public static Stream mk (Stream s) {
    return new Sieve(s);
  }

  public boolean hasElement () { return true; }

  public Integer current () { return this.str.current(); }

  public FuncIterator<Integer> advance () {
   return Sieve.mk(Stream.filter(NotDivByPredicate.mk(this.str.current()),
                   Stream.tail(this.str)));
  }
}
```

And indeed, executing

```
Stream primes = Sieve.mk(Stream.intsFrom(2));

printFirstNFromIterator(primes,14);
```

yields

```
2
3
5
7
11
13
```

```
17
19
23
29
31
37
41
43
```

Note, however, that this is far from being an efficient way for computing prime numbers, as you can tell immediately by running the above code.