30-Collections
Lists, etc.

Let’s talk about collections

- Real-life software generally involves <u>collections</u>.
  - Sometimes, we want to deal with one thing a time, sometimes many things at once.
  - These are the kinds of things we want to do with collections:
    - find the number of elements in the collection (*length/size*);
    - (optionally) select a specific element by position or key (*get*);
    - traverse each element in succession, applying a side-effect function to each (*foreach*);
    - traverse each element in succession, applying a function to each, thus yielding a new collection (*map, flatMap*);
    - traverse each element in succession, applying a function to each element and an accumulator, thus yielding a value (*reduce*);
    - create a new collection based on the original, but with only those elements that satisfy a predicate (*filter*);
    - create a new collection based on the original, but with new element(s) (concatenate or “*cons*”).

A simple definition of List

- This is (more or less) what I wrote on the board last time:

```
package edu.neu.coe.scala.list
trait List {
  def length: Int
}
case object Nil extends List {
  def length: Int = 0
}
case class Cons (head: Int, tail: List) extends List {
  def length: Int = this match {
   case Cons(h,t) => 1 + t.length
  }
}
object List {
 def apply(as: Int*): List =
    if (as.isEmpty) Nil
    else Cons(as.head, apply(as.tail: _*))
}
```

First, we define some behavior in a trait. For now, the only behavior we've defined for our *List* is the ability to get its length.

Remember *proof by induction*? There are two cases: the "base case" and the "inductive step". We will typically (but not always) have two "case class/object" extensions of a trait.

This is the "companion" object for *List*. Any class or trait can have such a companion object. Case classes always have one (via "syntactic sugar")

Parametric Types

- A very quick observation before we get into lists.
- We can define a *List* of *Int*, a *List* of *String*, etc.
- But we'd end up having to write all the same methods for each! That would be no good!!
- So, in Scala, all containers have an *underlying* type, including *List*. Such types are known as **Parametric types*** because that's what they are. Scala doesn't really use the term *generics* (partly because it wasn't an afterthought).
- Unlike in Java, we cannot define a *List* without a parametric type because this would break type inference.

* such types make up the *parameters* of a type—and are enclosed in [], just like parameters of a method are enclosed in ().

Lists and their methods

- Recap:

We have called the parametric type of the *List* “A”. A stands for any type, even a *List[[B]*. I will explain the “+” shortly.

```
package edu.neu.coe.scala.list
trait List[+A]
case object Nil extends List[Nothing]
case class Cons[+A] (head: A, tail: List[A]) extends List[A]
object List {
 def apply[A](as: A*): List[A] =
    if (as.isEmpty) Nil
    else Cons(as.head, apply(as.tail: _*))
}
```

The name for *Cons* in the Scala library is “::”

- What are the methods that we expect *List* to implement?
    - Let’s try a few signatures and think what they might mean:
        - def x0: Boolean
        - def x1: Int
        - def x2: A
        - def x3: List[A]
        - def x4(x: Int): Option[A]
        - def x5(f: A=>Boolean): List[A]
        - def x6(f: A=>Boolean): Option[A]
        - def x7[B](f: A=>B): List[B]
        - def x8[B](f: A=>List[B]): List[B]
        - def x9(f: A=>Unit): Unit

actually, there are a couple of plausible methods which yield an *A*

we are defining an “algebra” for the *List* type

List methods ("SOE")

```
def x0: Boolean = this match {case MyNil => true; case _ => false }

def x1: Int = this match {
  case Nil => 0
  case Cons(hd, tl) => 1 + tl.x1
}

def x2a: A = this match {
    case Nil => throw new Exception("logic error”)
    case Cons(hd,tl) => hd
  }

// Alternative interpretation
def x2b: A = this match {
  case Nil => unit(0)
  case Cons(hd, tl) => hd + tl.x2
}

def x3: List[A] = this match {
  case Nil => Nil;
  case Cons(hd, tl) => tl
}

def x4(x: Int): Option[A] = {
  @tailrec def inner(as: List[A], x: Int): Option[A] = as match {
    case Nil => None
    case Cons(hd, tl) => if (x == 0) Some(hd) else inner(tl, x - 1)
  }
  if (x < 0) None else inner(this, x)
}
```

Will not compile: need an operator + that can combine two *A* objects into another *A*. Need *unit* function, too.

List methods (higher-order functions)

- ```
  def x5(f: A=>Boolean): List[A] = this match {
    case Cons(hd,tl) => val ftl = tl.x5(f); if (f(hd)) Cons(hd, ftl) else ftl
    case Nil => Nil
  }
  ```
- ```
  def x6(f: A=>Boolean): Option[A] = this match {
    case Cons(hd,tl) => if (f(hd)) Some(hd) else tl.x6(f)
    case Nil => None
  }
  ```
- ```
  def x7[B](f: A=>B): List[B] = this match {
    case Cons(hd,tl) => Cons(f(hd),tl.x7(f))
    case Nil => List[B]()
  }
  ```
- ```
  def ++[B >: A](x: List[B]): List[B] = this match {
    case Nil => x
    case Cons(hd,tl) => Cons(hd, tl ++ x)
  }
  ```
- ```
  def x8[B](f: A=>List[B]): List[B] = this match {
     case Cons(hd,tl) => f(hd) ++ tl.x8(f)
     case Nil => List[B]()
  }
  ```
- ```
  def x9(f: A=>Unit): Unit = this match {
     case Cons(hd,tl) => f(hd); tl.x9(f)
     case Nil => Unit
  }
  ```

We need a way to concatenate two lists.

But can't we just use *A* as the type of both lists? No: co-/contra-variance

Giving the methods names:

- ```
  def isEmpty: Boolean = this match {case MyNil => true; case _ => false }
  ```

- ```
  def length: Int = this match {
    case Nil => 0
    case Cons(hd, tl) => 1 + tl.length
  }
  ```

- ```
  def head: A = this match {
    case Nil => throw new Exception("logic error”)
    case Cons(hd,tl) => hd
  }
  ```

- ```
  def sum: A = this match {
    case Nil => unit(0)
    case Cons(hd, tl) => hd + tl.sum
  }
  ```

  Will not compile: need an operator + that can combine two *A* objects into another *A*. Need *unit* function, too.

- ```
  def tail: List[A] = this match {
    case Nil => Nil;
    case Cons(hd, tl) => tl
  }
  ```

- ```
  def get(x: Int): Option[A] = {
    @tailrec def inner(as: List[A], x: Int): Option[A] = as match {
      case Nil => None
      case Cons(hd, tl) => if (x == 0) Some(hd) else inner(tl, x - 1)
    }
    if (x < 0) None else inner(this, x)
  }
  ```

and names for the higher-order functions…

- ```
  def filter(f: A=>Boolean): List[A] = this match {
    case Cons(hd,tl) => val ftl = tl.filter(f); if (f(hd)) Cons(hd, ftl) else ftl
    case Nil => Nil
  }
  ```
- ```
  def find(f: A=>Boolean): Option[A] = this match {
    case Cons(hd,tl) => if (f(hd)) Some(hd) else tl.find(f)
    case Nil => None
  }
  ```
- ```
  def map[B](f: A=>B): List[B] = this match {
    case Cons(hd,tl) => Cons(f(hd),tl.map(f))
    case Nil => List[B]()
  }
  ```
- ```
  def ++[B >: A](x: List[B]): List[B] = this match {
    case Nil => x
    case Cons(hd,tl) => Cons(hd, tl ++ x)
  }
  ```
- ```
  def flatMap[B](f: A=>List[B]): List[B] = this match {
     case Cons(hd,tl) => f(hd) ++ tl.flatMap(f)
     case Nil => List[B]()
  }
  ```
- ```
  def foreach(f: A=>Unit): Unit = this match {
     case Cons(hd,tl) => f(hd); tl.foreach(f)
     case Nil => Unit
  }
  ```

We need a way to concatenate two lists.

But can't we just use *A* as the type of both lists? No: co-/contra-variance

Method types

- (Refer to my StackOverflow answer for the original)
  - Let's assume a type *Bunch[T]* which extends *Traversable[T]* (the base trait for all Scala collections)
  - In the following, *U* is a supertype of *T:*
    - *traversing*: there are actually two subclasses:
      - *shape-preserving*: these define a return type of *Bunch[U]*; example: *map*;
      - *non-shape-preserving*: these define a return type of *Iterator[T]*, *Traversable[T]*, *Traversable[U]*, etc.; example: *iterator*;
    - *selecting*: these define a return type of *T*; example *head*;
    - *maybeSelecting*: these define a return type of *Option[U]*
    - *aggregation*: these define a return type of *U*; example: *foldLeft*;
    - *testing*: these define a return type of *Boolean*; example: *empty*;
    - *side-effecting*: these define a return type of *Unit*; example *foreach*;
    - etc. etc.