Linked lists and the List class

Otfried Cheong

1 Linked lists

So far, the only *container object* we used was the array. An array is a *single object* that contains references to other objects, possibly many of them. Its disadvantage is that its size is fixed. It is expensive to extend an array, to insert elements in the middle, or to remove an element from the sequence.

A *linked list* is a container that consists of many small objects called *nodes* that are linked together. Here is a `Node` class for a list of `String` objects:

```
class Node(val head:String,val tail:Node)
```

Note that this is a *recursive definition*: A `Node` contains a reference to an element (called `head`, a `String` object), and a reference to another `Node` object, the `tail` of the list.

Here is how we could create a short list with three strings:

```
scala> var l = new Node("apples", null)
scala> l = new Node("oranges", l)
scala> l = new Node("strawberries", l)
scala> l.head
res1: String = strawberries
scala> l.tail.head
res2: String = oranges
scala> l.tail.tail.head
res3: String = apples
scala> l.tail.tail.tail
res4: Node = null
```

Here is a function that will display all the elements of a list:

```
def displayList(L: Node) {
  if (L != null) {
    println(L.head)
    displayList(L.tail)
  }
}
```

Since `Node` was defined recursively, it is natural to display the list using a recursive function. Here is the output:

```
scala> displayList(l)
strawberries
oranges
apples
```

To create interesting lists, let us write a function that will read lines from a file and put them in a list:

```
def readWords(fname: String): Node = {
  val F = Source.fromFile(fname)
  var list: Node = null
  for (line <- F.getLines()) {
    list = new Node(line, list)
  }
  list
}
```

Note that the list will contain the lines of the files *in reverse order*, since every line is added to the *beginning* of the list. It is possible to build a list in the correct order, but that's a topic for later.

Lists should have some useful methods, so let us add a `length` method and a `display` method to the `Node` class:

```
class Node(val head:String,val tail:Node) {
  def length: Int = {
    if (tail == null) 1
    else 1 + tail.length
  }
  def display() {
    println(head)
    if (tail != null) tail.display()
  }
}
```

We can now determine the length of a list `L` as `L.length`, and display it using `L.display()`.

There are two problems with this setup: First, we represent the empty list using the `null` value. But that means that `L.length` and `L.display()` will throw a `NullPointerException` when `L` is an empty list! Second, it turns out that for long lists, both `L.length` and `L.display()` will cause a stack overflow, because the recursion becomes too deep

(the depth of the recursion is the same as the length of the list).

We can solve the second problem by rewriting the two methods using a loop, without recursion:

```
class Node(val head:String,val tail:Node) {
  def length: Int = {
    var p = tail
    var count = 1
    while (p != null) {
      count += 1
      p = p.tail
    }
    count
  }
  def display() {
    var p = this
    while (p != null) {
      println(p.head)
      p = p.tail
    }
  }
}
```

2 An empty list object

For the first problem, we need to give up the use of the `null` value. We create a singleton object that we can use to represent the empty list:

```
object Empty {
  def length: Int = 0
  def display() = ()
}
```

But now we have a problem: What is the type of a list variable? A list variable could either contain a reference to the `Empty` singleton, or a reference to a `Node` object. Those have two different types!

The solution is to create a `List` trait:

```
trait List {
  def head: String
  def tail: List
  def length: Int
  def display(): Unit
}
```

This trait shows that any `List` object must provide methods to access the head, the tail, and the length of the list, and must be able to display the list.

The `Empty` singleton and the `Node` class implement the `List` trait. This is indicated using the `extends` keyword:

```
object Empty extends List {
  def length: Int = 0
  def head: String =
    throw new NoSuchElementException
  def tail: List =
    throw new NoSuchElementException
  def display() = ()
}

class Node(val head: String,
           val tail: List) extends List {
  def length: Int = 1 + tail.length
  def display() {
    println(head)
    tail.display()
  }
}
```

Note how much simpler the two methods have become, since we do not need any special treatment for the empty list.

Our `readWords` function now needs only a small change: We need to initialize the variable `list` to the empty list `Empty`:

```
def readWords(fname: String): List = {
  val F = Source.fromFile(fname)
  var list: List = Empty
  for (line <- F.getLines()) {
    list = new Node(line, list)
  }
  list
}
```

Note that we really have to indicate the type `List` of the `list` variable. If we omit it, Scala will infer that `list` has type `Empty`, and we will not be able to assign a `Node` object to `list`.

3 Type parameters

This section contains a tricky point about covariance of type parameters. This is not required learning for the course and will not be on the exam!

As for our `GrowArray`, we can add a type parameter to the `List` trait so that we can have lists of any kind of object. What makes this a bit more complicated is the fact that a singleton object cannot have a type parameter. We therefore need to use the *same* `Empty` object for lists of all types. So what `List` trait should `Empty` implement? It turns out the the right answer is `List[Nothing]`:

```
trait List[+T] {
  def length: Int
  def head: T
  def tail: List[T]
  def display(): Unit
}

object Empty extends List[Nothing] {
  def length: Int = 0
  def head: Nothing =
    throw new NoSuchElementException
  def tail: List[Nothing] =
    throw new NoSuchElementException
  def display() = ()
}

class Node[T](val head: T,
    val tail: List[T]) extends List[T] {
  def length: Int // as before
  def display()   // as before
}
```

When we initialize a list variable to the empty list, we write code like this:

```
var L: List[String] = Empty
```

Even though the object `Empty` is of type `List[Nothing]`, it is legal to assign it to a variable of type `List[String]`. This is true because (a) `Nothing` is a *subtype* of every type, in particular of `String`, and (b) the `+T` in the type parameter of `List` makes `List` *covariant* in the type parameter `T`. This means that if `X` is a subtype of `Y`, then `List[X]` is a subtype of `List[Y]`. So as a result, `List[Nothing]` is a subtype of `List[String]`.

What is a subtype? A type `X` can be declared a subtype of a type `Y` if it implements all the methods of type `Y` in a compatible way. For instance, if `X` is a class that implements a trait `Y`, then `X` is a subtype of `Y`. When `X` is a subtype of `Y`, then it is legal to assign objects of type `X` to a variable of type `Y`.

4 The Scala List class

Fortunately, we don't need to worry about defining our own list class and its type parameters, because Scala already provides a nice List class that is very much like the class we have just created. In particular, it represents an *immutable* list. Here is how we can rewrite our `readWords` function using Scala `List`:

```
def readWords(fn:String): List[String] ={
  val F = Source.fromFile(fn)
  var list: List[String] = Nil
  for (line <- F.getLines()) {
    list = line :: list
  }
  list.reverse
}
```

The name of the empty list singleton is `Nil` (as in Scheme). We construct nodes of the list using the `::` operator:

```
scala> val A = 1 :: Nil
A: List[Int] = List(1)
scala> val B = 1 :: 2 :: 3 :: Nil
B: List[Int] = List(1, 2, 3)
scala> val C = "Otfried" :: "Jungwoo" ::
       "Youngwoon" :: Nil
C: List[String] =
   List(Otfried, Jungwoo, Youngwoon)
```

The `::` operator is pronounced *cons*, just as in Scheme.

As you can see, Scala displays the list `1 :: 2 :: Nil` as `List(1,2)`. You can create lists with this syntax as well (similar to arrays):

```
scala> val D = List('a','b','c')
D: List[Char] = List(a, b, c)
```

In fact, to create an empty list it is better to write `List()` instead of `Nil`, since it gives the type we usually want:

```
scala> var s = Nil
s: object Nil = List()
scala> var t = List()
t: List[Nothing] = List()
```

As in our own list, the first element of a list is its `head`, the remainder of the list is its `tail`:

```
scala> C.head
res1: String = Otfried
scala> C.tail
res2: List[String] =
      List(Jungwoo, Youngwoon)
```

The operation `:::` concatenates two lists:

```
scala> B ::: (5 :: A)
res3: List[Int] = List(1, 2, 3, 5, 1)
```

You can convert lists to arrays, and vice versa:

```
scala> D.toArray
res4: Array[Char] = Array(a, b, c)
scala> val E = Array(1, 2, 3, 4)
E: Array[Int] = Array(1, 2, 3, 4)
scala> E.toList
res5: List[Int] = List(1, 2, 3, 4)
```

Lists have many methods in common with arrays, for instance (`L` is a list):

- `L.length`
- `L.isEmpty`
- `L.nonEmpty`
- `L drop n`
- `L dropRight n`
- `L take n`
- `L splitAt n`, which returns a pair consisting of `L take n` and `L drop n`
- `L.mkString`
- `L.mkString(separator)`
- `L.mkString(prefix, separator, suffix)`
- `L.reverse`
- `L.sorted`

5 Pattern matching

The `::` operator can also be used as a *pattern*. For instance, we can decompose a list into its first two elements and the rest of the list:

```
scala> val F = List(1, 2, 3, 4, 5, 6)
scala> val el1 :: el2 :: rest = F
el1: Int = 1
el2: Int = 2
rest: List[Int] = List(3, 4, 5, 6)
```

If the pattern does not match, an exception is thrown:

```
scala> val G = List('a')
G: List[Char] = List(a)
scala> val el1 :: el2 :: rest = G
scala.MatchError: List(a)
```

Patterns can also be used inside the cases of a `match` block:

```
def length(L: List[String]): Int = {
  L match {
    case Nil => 0
    case el :: rest => 1 + length(rest)
  }
}
```

```
def display(L: List[String]) {
  L match {
    case Nil =>
    case el :: rest =>
      println(el); display(rest)
  }
}
```

Note how the pattern matching here nicely takes care of both distinguishing the two cases and extracting the head and tail of the list.

Let us study some more examples. Here is a function `take(L, n)`, that does the same as `L take n`:

```
def take(L: List[String], n: Int):
                          List[String] = {
  if (n <= 0)
    Nil
  else
    L match {
      case Nil => Nil
      case x :: xs => x :: take(xs, n-1)
    }
}
```

And the same for `drop(L, n)`:

```
def drop(L: List[String], n: Int):
                          List[String] = {
  if (n <= 0)
    L
  else
    L match {
      case Nil => Nil
      case x :: xs => drop(xs, n-1)
    }
}
```

What about concatenating two lists? Again we can express it recursively:

```
def concat(L1: List[String],
           L2: List[String]):
           List[String] = {
  L1 match {
    case Nil => L2
    case x :: xs => x :: concat(xs, L2)
  }
}
```

And now for something more interesting: Assume we have a list that is sorted in increasing order, and we want to insert a new element into this list. Again, we can write this as a recursive function:

```
def insert(L: List[String], x: String):
           List[String] = {
  L match {
    case Nil => List(x)
    case y::ys => if (x < y) x :: L
                  else y :: insert(ys, x)
  }
}
```

But this means we can now sort lists!

```
def sort(L:List[String]):List[String] = {
  L match {
    case Nil => Nil
    case x :: xs => insert(sort(xs), x)
  }
}
```

6 Function objects

We have seen how to define functions and methods using the `def` keyword. Such a method is compiled to some code with the given name.

In Scala, it is also possible to create *function objects* without giving it a name. A function object is an object that can be used like a function (that is, it has an `apply` method).

Here is a simple example:

```
scala> (x: Int) => x + 1
res1: (Int) => Int = <function1>
```

Here we have created a function object that takes one `Int` argument and returns the argument plus one. The type of the object is `(Int) => Int`. A function object can be used like a function:

```
scala> val f = (x: Int) => x + 1
f: (Int) => Int = <function1>
scala> f(3)
res1: Int = 4
scala> f(7)
res2: Int = 8
scala> f(9)
res3: Int = 10
```

We can make this even more interesting. In the following function object `g`, the function definition makes use of a variable `more`:

```
scala> var more = 5
more: Int = 5
scala> val g = (x: Int) => x + more
g: (Int) => Int = <function1>
scala> g(4)
res1: Int = 9
scala> g(10)
res2: Int = 15
```

But what happens if we change the value of `more`? The answer is that the behavior of the function object changes as well:

```
scala> more = 10
more: Int = 10
scala> g(4)
res3: Int = 14
scala> g(10)
res4: Int = 20
```

We say that the variable `more` is a *free variable* of the function object. The behavior of a function object depends not only on its arguments, but also on its free variables.

7 Higher-Order methods

When working with lists, there are many common functions that can be implemented as a `for`-loop:

```
scala> for (e <- C)
     |   println(e)
Otfried
Jungwoo
Youngwoon
```

Higher-order methods allow us to concentrate on the interesting part of this loop, namely the print statement:

```
scala> C.foreach((x: String) => println(x))
Otfried
Jungwoo
Youngwoon
```

The `foreach` method is called a *higher-order* method because its argument is itself a function object. In a sense, `foreach` is a "meta-function" that works on other functions.

The code above can be simplified, because the Scala compiler knows that the argument of `foreach` has to be a function object. Therefore we are allowed to omit the type of the argument:

```
scala> C.foreach((x) => println(x))
Otfried
Jungwoo
Youngwoon
```

In this case, there is only one argument, and so we are even allowed to remove the parentheses:

```
scala> C.foreach(x => println(x))
Otfried
Jungwoo
Youngwoon
```

As a final simplification, when the function object has only a single argument and this argument is only used once in the result expression, we can omit the `x =>` part completely and replace the parameter by an underscore:

```
scala> C.foreach(println(_))
Otfried
Jungwoo
Youngwoon
```

Many operations on list that would normally use a `for`-loop can be written using `foreach`. For instance, we can compute the sum of all elements in a list like this:

```
scala> B
res1: List[Int] = List(1, 2, 3)
scala> var sum = 0
sum: Int = 0
scala> B.foreach(sum += _)
scala> sum
res2: Int = 6
```

Here are the most important higher-order methods of lists:

- `L.foreach(f)` calls `f(e)` for each element `e` of `L`;
- `L.exists(f)` returns true if for *some element* `e` of `L` the function `f(e)` returns true;
- `L.forall(f)` returns true if for *all elements* `e` of `L` the function `f(e)` returns true;
- `L.count(f)` returns the number of elements `e` of `L` for which the function `f(e)` returns true;
- `L.filter(f)` returns a list consisting of those elements `e` of `L` for which the function `f(e)` returns true;
- `L.filterNot(f)` is the same as `L.filter(e => !f(e))`.
- `L.map(f)` returns a new list containing the elements `f(a), f(b),...`, where `a,b,...` are the elements of `L`.
- `L.sortWith(f)` sorts the list using `f` as the comparison function. `f(a,b)` takes two list elements and returns true if `a` should come before `b` in the sorted order.

For instance:

```
scala> val words =
  Source.fromFile("words.txt").
    getLines().toList
words = List(aa, aah, aahed, ...)
scala> words filter (_ contains "issis")
res1 = List(missis, missises, narcissism,
narcissisms, narcissist, narcissists)
scala> words count (_.length > 20)
res2: Int = 3
scala> words exists (_ startsWith "kw")
res3: Boolean = true
```

As a final example, here is a program to compute prime numbers:

```
val n = args(0).toInt
val sqrtn = math.sqrt(n).toInt

var s = (2 to n).toList

while (s.head <= sqrtn) {
  print(s.head + " ")
  s = s.tail filter (_ % s.head != 0)
}
println(s.mkString(" "))
```

In Scala, higher-order methods are not only available for lists, but also for arrays, strings, and in fact any sequence.