

CompSci 220

Programming Methodology

12: Functional Data Structures

A Polymorphic Higher-Order Function

```
def findFirst[A](as: Array[A], p: A => Boolean): Int = {  
  def loop(n: Int): Int =  
    if (n >= as.length) -1  
    else if (p(as(n))) n  
    else loop(n + 1)  
  
  loop(0)  
}
```

A Polymorphic Higher-Order Function

```
def findFirst[A](as: Array[A], p: A => Boolean): Int = {  
  def loop(n: Int): Int =  
    if (n >= as.length) -1  
    else if (p(as(n))) n  
    else loop(n + 1)  
  
  loop(0)  
}
```

The findFirst function is polymorphic
in its type parameter A.

A Polymorphic Higher-Order Function

```
def findFirst[A](as: Array[A], p: A => Boolean): Int = {  
  def loop(n: Int): Int =  
    if (n >= as.length) -1  
    else if (p(as(n))) n  
    else loop(n + 1)  
  
  loop(0)  
}
```

The type parameter is used to restrict the type of the elements contained in the Array to A.

A Polymorphic Higher-Order Function

```
def findFirst[A](as: Array[A], p: A => Boolean): Int = {  
  def loop(n: Int): Int =  
    if (n >= as.length) -1  
    else if (p(as(n))) n  
    else loop(n + 1)  
  
  loop(0)  
}
```

And to restrict the type of the argument of the *predicate* function **p** to **A**.

We use the function **p** to abstract the task of determining what we are looking for.

A Polymorphic Higher-Order Function

```
def findFirst[A](as: Array[A], p: A => Boolean): Int = {  
  def loop(n: Int): Int =  
    if (n >= as.length) -1  
    else if (p(as(n))) n  
    else loop(n + 1)  
  
  loop(0)  
}
```

loop performs the iteration over the array of elements of type A.

This is a tail-recursive function – the Scala compiler will translate this into an efficient loop!

A Polymorphic Higher-Order Function

```
def findFirst[A](as: Array[A], p: A => Boolean): Int = {  
  def loop(n: Int): Int =  
    if (n >= as.length) -1  
    else if (p(as(n))) n  
    else loop(n + 1)  
  
  loop(0)  
}
```

We invoke **p** on each element in the array to determine if we found the item.

Because the types match up – this expression is *type safe*.

Implement a Higher-Order Function

```
def isSorted[A](as: Array[A], ord: (A,A) => Boolean): Boolean = {  
  
  // Implement this on a piece of paper  
  // Hand in this paper at the end of class  
  
}
```


Implement a Higher-Order Function

```
def isSorted[A](as: Array[A], ord: (A,A) => Boolean): Boolean = {  
  def loop(a: Array[A], res: Boolean): Boolean =  
    if (a.isEmpty) res && true  
    else if (a.length == 1) res && true  
    else loop(a.tail, ord(a(0),a(1)))  
  
  loop(as, true)  
}
```

Here, we create a tail-recursive **loop** function that traverses the array.

We perform the proper case analysis.

We use the **ord** function to define the ordering we are interested in.

Anonymous Functions

```
def isSorted[A](as: Array[A], ord: (A,A) => Boolean): Boolean = {  
  def loop(a: Array[A], res: Boolean): Boolean =  
    if (a.isEmpty) res && true  
    else if (a.length == 1) res && true  
    else loop(a.tail, ord(a(0),a(1)))  
  
  loop(as, true)  
}
```

We use an *anonymous function* to invoke the **isSorted** function.

This is also called a *lambda*.

```
isSorted(Array(1,2,3,4), (x: Int,y: Int) => x <= y)
```

See Sorted01.scala

Short Definition, Many Possibilities

- The **isSorted** function is a short function.
 - It is parameterized by type *A* to allow for any type of *Array[A]*.
 - It is parameterized by a function **ord** to allow for any ordering definition.
-
- This allows for many possible uses of the function **isSorted**.

Short Definition, One Possibility

- Type and function parameterization often allows for many possible uses.
- It can also be used to constrain the implementation to only one possibility.

Consider the higher-order function for *partial application*. It takes a value and a function of two arguments, and returns a function of one argument as its result:

```
def partial1[A,B,C](a: A, f: (A,B) => C): B => C = ???
```

Short Definition, One Possibility

- How would you go about implementing this higher-order function?
- It turns out that there is only one implementation that works and it follows logically from the type signature of the **partial1** function.

```
def partial1[A,B,C](a: A, f: (A,B) => C): B => C = ???
```

Short Definition, One Possibility

- How would you go about implementing this higher-order function?
- It turns out that there is only one implementation that works and it follows logically from the type signature of the **partial1** function.
- We know that we need to return a function.

```
def partial1[A,B,C](a: A, f: (A,B) => C): B => C = ???
```

Short Definition, One Possibility

- How would you go about implementing this higher-order function?
- It turns out that there is only one implementation that works and it follows logically from the type signature of the **partial1** function.
- We know that we need to return a function.
- We also know that that function takes a parameter of type *B*.

```
def partial1[A,B,C](a: A, f: (A,B) => C): B => C = ???
```

Short Definition, One Possibility

- How would you go about implementing this higher-order function?
- It turns out that there is only one implementation that works and it follows logically from the type signature of the **partial1** function.
- We know that we need to return a function.
- We also know that that function takes a parameter of type *B*.
- So, we can easily start off the implementation.

```
def partial1[A,B,C](a: A, f: (A,B) => C): B => C =  
  (b: B) => ???
```


Short Definition, One Possibility

- How would you go about implementing this higher-order function?
- It turns out that there is only one implementation that works and it follows logically from the type signature of the **partial1** function.
- We know that we need to return a function.
- We also know that that function takes a parameter of type *B*.
- So, we can easily start off the implementation.
- What is the implementation?

```
def partial1[A,B,C](a: A, f: (A,B) => C): B => C =  
  (b: B) => ???
```

Short Definition, One Possibility

- How would you go about implementing this higher-order function?
- It turns out that there is only one implementation that works and it follows logically from the type signature of the **partial1** function.
- We know that we need to return a function.
- We also know that that function takes a parameter of type *B*.
- So, we can easily start off the implementation.
- What is the implementation?

```
def partial1[A,B,C](a: A, f: (A,B) => C): B => C =  
  (b: B) => f(a, b)
```

See Partial02.scala

How about this function?

The **curry** function converts a function f of two arguments into a function of one argument that partially applies f .

Again, there is only one implementation of this.

```
def curry[A,B,C](f: (A,B) => C): A => (B => C) = ???
```

How about this function?

The **curry** function converts a function f of two arguments into a function of one argument that partially applies f .

Again, there is only one implementation of this.

```
def curry[A,B,C](f: (A,B) => C): A => (B => C) =  
  (a: A) => (b: B) => f(a,b)
```

See Curry03.scala

The real world?

- These functions are interesting, but how do they relate to real-world programming in the large?

The real world?

- These functions are interesting, but how do they relate to real-world programming in the large?
- It turns out by using functions such as *partial* and *curry*, as well as many others (e.g., *compose*, *map*, *fold*, *reduce*, *zip*, ...) you are able to construct sophisticated programs using functional composition.

The real world?

- These functions are interesting, but how do they relate to real-world programming in the large?
- It turns out by using functions such as *partial* and *curry*, as well as many others (e.g., *compose*, *map*, *fold*, *reduce*, *zip*, ...) you are able to construct sophisticated programs using functional composition.
- Polymorphic, higher-order functions often end up being extremely widely applicable, precisely because they say nothing about any particular domain and are simply abstracting over common **patterns** that occur in many contexts.

What about data structures?

- Using pure functions lends itself naturally to *immutable* data structures. Pure functions do not change state and immutable data structures can't be modified.
- How might we define a List data structure?

See List04.scala

List Abstract Data Type

- How do we represent a List such that it is immutable?

List Abstract Data Type

- How do we represent a List such that it is immutable?
 - We need something that represents the empty list.

List Abstract Data Type

- How do we represent a List such that it is immutable?
 - We need something that represents the empty list.
 - We need something that represents a non-empty list

List Abstract Data Type

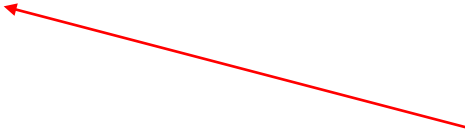
- How do we represent a List such that it is immutable?
 - We need something that represents the empty list.
 - We need something that represents a non-empty list

```
sealed trait List[+A]
```

List Abstract Data Type

- How do we represent a List such that it is immutable?
 - We need something that represents the empty list.
 - We need something that represents a non-empty list

sealed trait List[+A]



A trait that is “sealed” simply means that anything that extends it must be defined in the same file.

List Abstract Data Type

- How do we represent a List such that it is immutable?
 - We need something that represents the empty list.
 - We need something that represents a non-empty list

sealed trait List[+A]




The + in front of the type parameter *A* is a *variance* annotation. In particular, the *+A* indicates that the type parameter *A* is *covariant*.

That is, if *X* is a subtype of *A* then List[*X*] is a subtype of List[*A*].

List Abstract Data Type

- How do we represent a List such that it is immutable?
 - We need something that represents the empty list.
 - We need something that represents a non-empty list

```
sealed trait List[+A]  
case object Nil extends List[Nothing]
```




Nil represents the empty list.
Nil is an *object* – there is only one of these. Why?

List Abstract Data Type

- How do we represent a List such that it is immutable?
 - We need something that represents the empty list.
 - We need something that represents a non-empty list

```
sealed trait List[+A]  
case object Nil extends List[Nothing]
```




Nothing is a subtype of everything. This means that Nil can be used in the context of any type of list: List[Int], List[String], etc.

List Abstract Data Type

- How do we represent a List such that it is immutable?
 - We need something that represents the empty list.
 - We need something that represents a non-empty list

```
sealed trait List[+A]  
case object Nil extends List[Nothing]  
case class Cons[+A](head: A, tail: List[A]) extends List[A]
```



Cons is a case class. We will represent a List as a head element (the data) and the rest of the list (the tail).

Constructing Lists

- So, lists are represented by a “cons cell”.
How do we create lists then?

```
val xs = Cons(1, Cons(2, Cons(3, Nil)))
```

Constructing Lists

- So, lists are represented by a “cons cell”.
How do we create lists then?

```
val xs = Cons(1, Cons(2, Cons(3, Nil)))
```

Isn't this rather ugly?

I would think so

How might we have our implementation support this: *List(1,2,3)*?

Constructing Lists

- So, lists are represented by a “cons cell”. How do we create lists then?

- a) Create a function called List
- b) Create a companion object with an apply method?
- c) Create an apply method in the List class?
- d) Use Scala’s List implementation to do this.
- e) None of these.

```
val xs = Cons(1, Cons(2, Cons(3, Nil)))
```

Isn’t this rather ugly?

I would think so

How might we have our implementation support this: *List(1,2,3)*?

The List Companion Object

- Companion objects are perfect for this sort of thing.

Can anyone suggest on how to implement this?
Go ahead, do not be shy 😊

The List Companion Object

- Companion objects are perfect for this sort of thing.

```
object List {  
  def apply[A](as: A*): List[A] =  
    if (as.isEmpty) Nil  
    else Cons(as.head, apply(as.tail: _*))  
}
```

Let us create a sum method in List

- How would you write a function List.sum that takes a List[Int] and returns the sum of each Int in the list?

```
sealed trait List[+A]  
case object Nil extends List[Nothing]  
case class Cons[+A](head: A, tail: List[A]) extends List[A]
```

```
object List {  
  def sum(xs: List[Int]): Int = ???  
}
```

Let us create a sum method in List

- How would you write a function List.sum that takes a List[Int] and returns the sum of each Int in the list?

```
sealed trait List[+A]  
case object Nil extends List[Nothing]  
case class Cons[+A](head: A, tail: List[A]) extends List[A]
```

```
object List {  
  def sum(xs: List[Int]): Int =  
    if (xs == Nil) 0  
    else xs.head + sum(xs.tail)  
}
```

Here is one way of doing it. But, it turns out there is a more elegant way of expressing this in Scala...

Let us create a sum method in List

- How would you write a function List.sum that takes a List[Int] and returns the sum of each Int in the list?

```
sealed trait List[+A]  
case object Nil extends List[Nothing]  
case class Cons[+A](head: A, tail: List[A]) extends List[A]
```

```
object List {  
  def sum(xs: List[Int]): Int = xs match {  
    case Nil => 0  
    case Cons(x, xs) => x + sum(xs)  
  }  
}
```

Pattern matching is the preferred approach to “match” over data structures.

Let us create a sum method in List

- How would you implement product?

```
sealed trait List[+A]  
case object Nil extends List[Nothing]  
case class Cons[+A](head: A, tail: List[A]) extends List[A]
```

```
object List {  
  def product(xs: List[Double]): Double = ???  
}
```

Let us create a sum method in List

- How would you implement product?

```
sealed trait List[+A]
case object Nil extends List[Nothing]
case class Cons[+A](head: A, tail: List[A]) extends List[A]
```

```
object List {
  def product(xs: List[Double]): Double = xs match {
    case Nil => 1.0
    case Cons(0.0, _) => 0.0
    case Cons(x, xs) => x * product(xs)
  }
}
```

The `_` pattern matches anything. In this case if we see a 0.0, then the rest of the list does not matter.

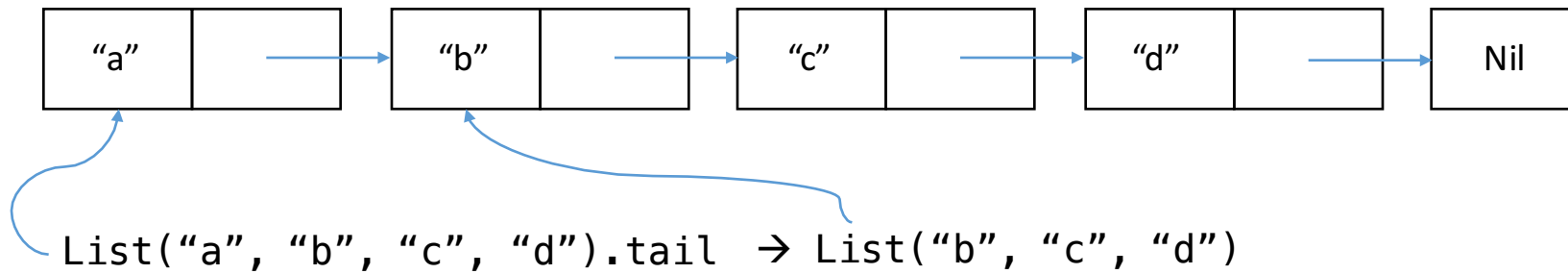
Data Sharing

- When data is immutable, how do we write functions that add or remove elements from a list?

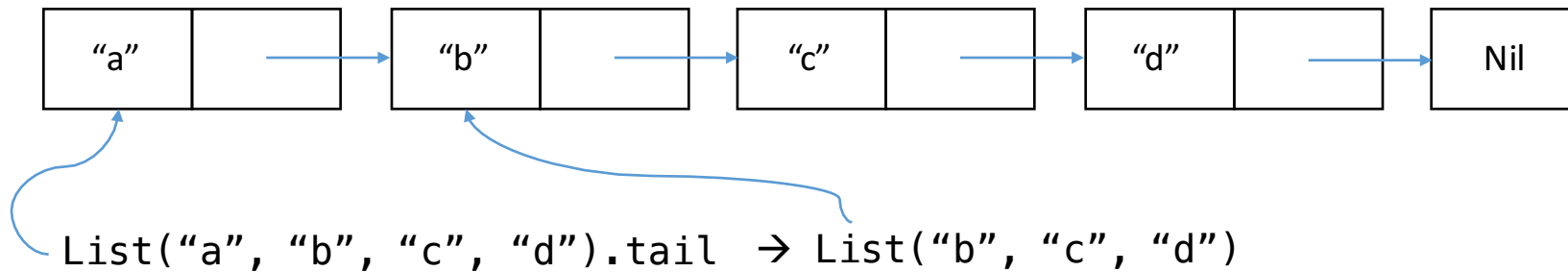
```
object List {  
  def add[A](xs: List[A], e: A): List[A] =  
    Cons(e, xs)  
}
```

- We need not copy `xs`, we simply reuse it.
- Sharing of immutable data simplifies code (copying unnecessary)
- Functional data structures are *persistent* meaning that existing references are never changed by operations on the data structure.

Data Structures are Persistent



Data Structures are Persistent



How would you implement the function `List.tail`?
See List04.scala

Generalizing to Higher-Order Functions

- Remember the `List.sum` and `List.product` methods?
- Is there a common pattern that we could abstract from both?

Generalizing to Higher-Order Functions

- Remember the `List.sum` and `List.product` methods?
- Is there a common pattern that we could abstract from both?
- Right, they both traverse a list to produce a single value.

- What if we could write a function that traverses a list to produce a single value and use that to implement `sum` and `product`?

This is the hallmark of function programming.

foldRight

- The foldRight function does exactly that – it traverses a list and produces a final value by combining elements in the list.

```
object List {  
  def foldRight[A,B](xs: List[A], z: B)(f: (A,B) => B): B =  
    xs match {  
      case Nil => z  
      case Cons(x, xs) => f(x, foldRight(xs, z)(f))  
    }  
}
```

Now, implement sum and product in terms of that...

foldRight

- The foldRight function does exactly that – it traverses a list and produces a final value by combining elements in the list.

```
object List {  
  def foldRight[A,B](xs: List[A], z: B)(f: (A,B) => B): B =  
    xs match {  
      case Nil => z  
      case Cons(x, xs) => f(x, foldRight(xs, z)(f))  
    }  
  
  def sum(xs: List[Int]) =  
    foldRight(xs, 0)((x,y) => x + y)  
}
```

foldRight *unfolded*

```
foldRight(Cons(1, Cons(2, Cons(3, Nil))), 0)((x,y) => x + y)
```

foldRight *unfolded*

```
foldRight(Cons(1, Cons(2, Cons(3, Nil))), 0)((x,y) => x + y)
```

```
1 + foldRight(Cons(2, Cons(3, Nil)), 0)((x,y) => x + y)
```

foldRight *unfolded*

```
foldRight(Cons(1, Cons(2, Cons(3, Nil))), 0)((x,y) => x + y)
```

```
1 + foldRight(Cons(2, Cons(3, Nil)), 0)((x,y) => x + y)
```

```
1 + (2 + foldRight(Cons(3, Nil), 0)((x,y) => x + y))
```

foldRight *unfolded*

```
foldRight(Cons(1, Cons(2, Cons(3, Nil))), 0)((x,y) => x + y)
```

```
1 + foldRight(Cons(2, Cons(3, Nil)), 0)((x,y) => x + y)
```

```
1 + (2 + foldRight(Cons(3, Nil), 0)((x,y) => x + y))
```

```
1 + (2 + (3 + foldRight(Nil, 0)((x,y) => x + y)))
```

foldRight *unfolded*

```
foldRight(Cons(1, Cons(2, Cons(3, Nil))), 0)((x,y) => x + y)
```

```
1 + foldRight(Cons(2, Cons(3, Nil)), 0)((x,y) => x + y)
```

```
1 + (2 + foldRight(Cons(3, Nil), 0)((x,y) => x + y))
```

```
1 + (2 + (3 + foldRight(Nil, 0)((x,y) => x + y)))
```

```
1 + (2 + (3 + (0)))
```

foldRight *unfolded*

```
foldRight(Cons(1, Cons(2, Cons(3, Nil))), 0)((x,y) => x + y)
```

```
1 + foldRight(Cons(2, Cons(3, Nil)), 0)((x,y) => x + y)
```

```
1 + (2 + foldRight(Cons(3, Nil), 0)((x,y) => x + y))
```

```
1 + (2 + (3 + foldRight(Nil, 0)((x,y) => x + y))
```

```
1 + (2 + (3 + (0)))
```

6