Comp 311 Functional Programming

Eric Allen, Two Sigma Investments Robert "Corky" Cartwright, Rice University Sagnak Tasirlar, Two Sigma Investments

Announcements

- Homework 2 Available from Piazza (Due October 6)
- Two Sigma Info Session at Huff House, 8pm Today

Additional Syntactic Forms

- Scala allows the last parameter to a function to stand for zero or more arguments
- The arguments are placed into an Array of the given type

- Scala allows the last parameter to a function to stand for zero or more arguments
- The arguments are placed into an Array of the given type

```
squares(4,2,6,5,8)
    squares()
    squares(4,2,6,8)
        squares(3)
    squares(4,3,7)
```

- Scala allows the last parameter to a function to stand for zero to many arguments
- The arguments are placed into an Array of the given type

def fnName(arg0, ..., argN: Type*) = expr

 If you have an array and you wish to pass it to a repeated parameter, include the suffix :_*

ArrayBuffers

- Buffers in Scala enable incremental creation of sequences
 - Support destructive append, prepend, insert
 - We have not talked about destructive operations yet
 - Just pretend they are arrays for now
 - Random access to elements
- ArrayBuffers are simply Buffers implemented using Arrays

 If you have an array and you wish to pass it to a repeated parameter, include the suffix :_*

Guidelines on Repeated Parameters

 Use repeated parameters to provide factory methods for collections classes

List(1,2,3,4,5)

 Use repeated parameters for methods that map over an immediately provided set of values

squares(1,2,3,4,5)

Use repeated parameters for folds over an immediately provided set of values

Named Arguments

- With *named arguments*, the arguments to a function can be passed in any order
- Each argument must be prefixed with the name of the parameter and an equals sign:

def speed(distance: Double, time: Double) =
 distance/time

speed(time = 5.0, distance = 2.0)

Named Arguments

 If positional arguments are mixed with named arguments, the positional arguments must come first

```
def speed(distance: Double, time: Double) =
   distance/time
```

speed(2.0, time = 5.0)

Guidelines on Named Arguments

- Named arguments add bulk to function applications
- Use when:
 - There are multiple arguments of the same type
 - It's important which arguments correspond to which parameters
 - There is no natural order for the arguments
 - The expected order of the arguments is difficult to remember

Default Parameter Values

Function parameters can include default values:

case class Circle(radius: Double = 1) extends Shape {
 val pi = 3.14

```
def area = { pi * radius * radius }
def makeLikeMe(that: Shape): Circle = this
```

}

• The argument for a parameter with a default value can be omitted at the call site:

Guidelines of Default Parameter Values

- Consider default parameter values instead of static overloading
- Use when there is a common argument value that is usually used
 - A default I/O source, file location, etc.

Imports

Importing a Member of a Package

import scala.collection.immutable.List

Importing Multiple Members of a Package

import scala.collection.immutable.{List, Vector}

Importing and Renaming Members of a Package

import scala.collection.immutable.{List=>SList, Vector}

Importing All Members of a Package

import scala.collection.immutable._

Note that * is a valid identifier in Scala!

Combining Notations

import scala.collection.immutable.{_}

same meaning as:

import scala.collection.immutable._

Combining Notations

import scala.collection.immutable.{List=>SList,_}

Imports all members of the package but renames List to SList

Combining Notations

import scala.collection.immutable.{List=>_,_}

Imports all members of the package except for List

Importing a Package

import scala.collection.immutable

Now sub-packages can be denoted by shorter names:

immutable.List

Importing and Renaming Packages

import scala.collection.{immutable => I}

Allows members to be written like this:

I.List

Importing Members of An Object

import Arithmetic._

Allows members such as **Arithmetic.gcd** to be write like this:

gcd

Implicit Imports

The following imports are implicitly included in your program:

import java.lang._
import scala._
import Predef._

Package java.lang

- Contains all the standard Java classes
- This import allows you to write things like:

Thread

instead of:

java.lang.Thread

Package scala

• Provides access to the standard Scala classes:

BigInt, BigDecimal, List, etc.

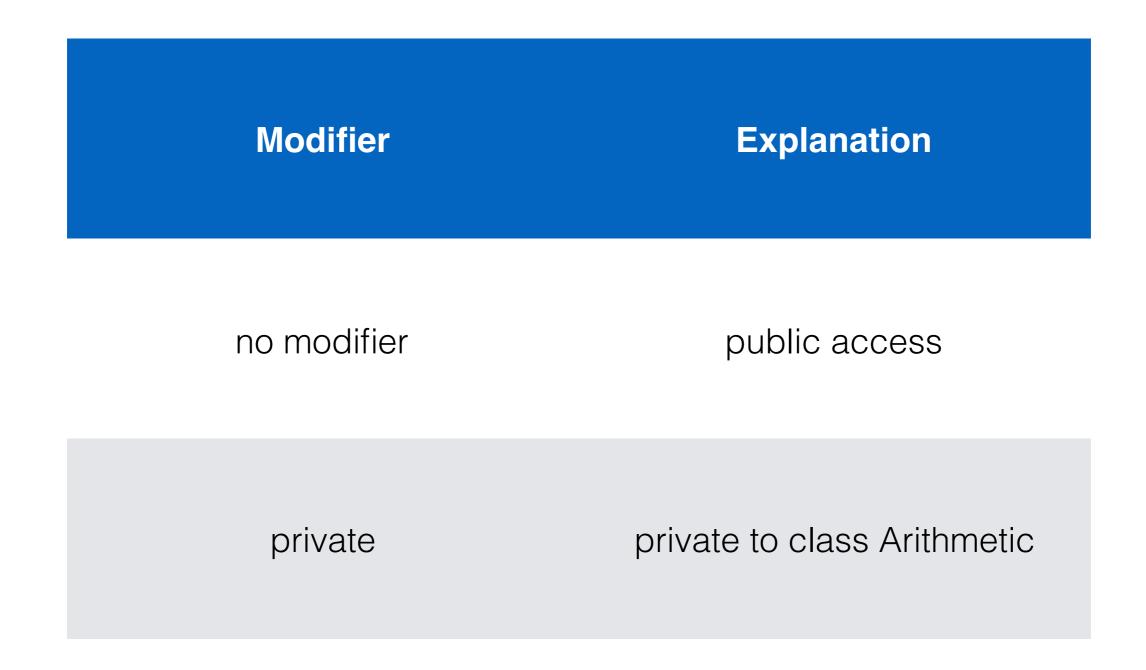
Object Predef

 Definitions of many commonly used types and methods, such as:

require, ensuring, assert

Visibility Modifier Private

For a method Arithmetic.reduce in package Rationals



Higher Order Functions

Comprehensions

$\{2x \mid x \in xs\}$

Mapping a Computation Over a List

```
def double(xs: List) = {
    xs match {
        case Empty => Empty
        case Cons(y,ys) => Cons(2 * y, double(ys))
    }
}
```

Mapping a Computation Over a List

```
def negate(xs: List) = {
    xs match {
        case Empty => Empty
        case Cons(y,ys) => Cons(-y, negate(ys))
    }
}
```

Negation as a Comprehension

 $\{-x \mid x \in xs\}$

Generalizing a Mapping Computation

```
def map(f: Int => Int, xs: List) = {
    xs match {
        case Empty => Empty
        case Cons(y,ys) => Cons(f(y), map(f,ys))
    }
}
```

Mapping a Computation Over a List

val xs = Cons(1,Cons(2,Cons(3,Cons(4,Cons(5,Cons(6,Empty))))))

negate(xs) →*
Cons(-1,Cons(-2,Cons(-3,Cons(-4,Cons(-5,Cons(-6,Empty)))))

double(xs) →*
Cons(1,Cons(4,Cons(9,Cons(16,Cons(25,Cons(36,Empty)))))

Mapping a Computation Over a List

val xs = Cons(1,Cons(2,Cons(3,Cons(4,Cons(5,Cons(6,Empty))))))

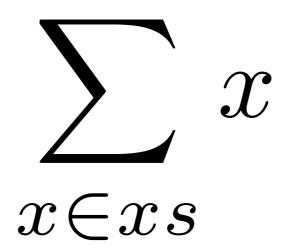
 $map(-_, xs) \mapsto^*$ Cons(-1,Cons(-2,Cons(-3,Cons(-4,Cons(-5,Cons(-6,Empty)))))

map(x => 2 * x, xs) →*
Cons(1,Cons(4,Cons(9,Cons(16,Cons(25,Cons(36,Empty))))))

Recall Our Sum Function Over Lists

def sum(xs: List): Int = {
 xs match {
 case Empty => 0
 case Cons(y,ys) => y + sum(ys)
 }
}

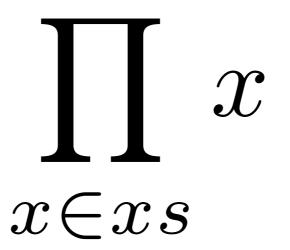
In Mathematics, We Might Write this as a Summation



And Our Product Function Over Lists

def product(xs: List): Int = {
 xs match {
 case Empty => 1
 case Cons(y,ys) => y * product(ys)
 }
}

In Mathematics, We Might Write this as a Product



We Abstract to a Reduction Function Over Lists

```
def reduce(base: Int, f: (Int, Int) => Int, xs: List): Int = {
    xs match {
        case Empty => base
        case Cons(y,ys) => f(y, reduce(base, f, ys))
    }
}
```

Example Reductions

val xs = Cons(1,Cons(2,Cons(3,Cons(4,Cons(5,Cons(6,Empty)))))

reduce(0, (x,y) => x + y, xs) \mapsto * 21

reduce(1, (x,y) => x * y, xs) \mapsto * 720

Min and Max

def max(xs: List) = {
 reduce(Int.MinValue, (x,y) => if (x > y) x else y, xs)
}

def min(xs: List) = {
 reduce(Int.MaxValue, (x,y) => if (x < y) x else y, xs)
}</pre>

Simplifying Function Literals

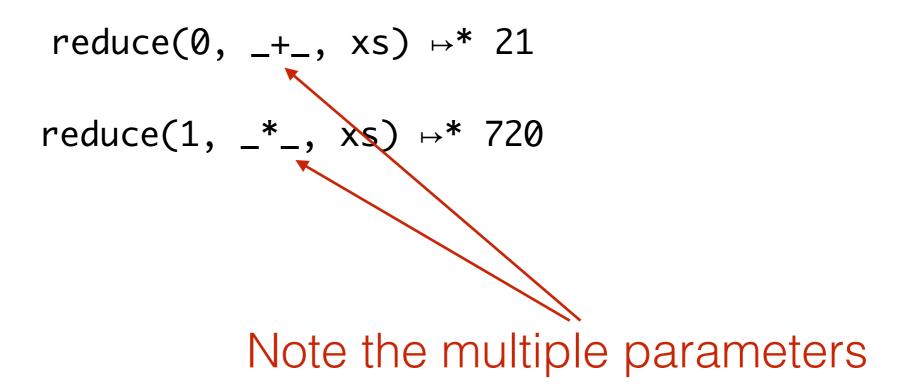
- When *each* parameter is used only once in the body of a function literal, and in the order in which they are passed:
 - We can drop the parameter list
 - We simply write the body with an _ at the place where each parameter is used

For example,

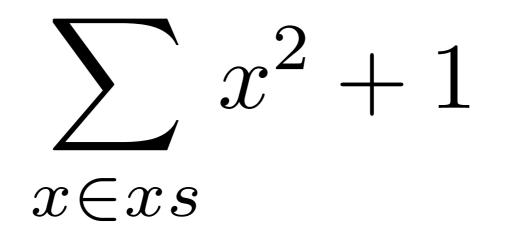
becomes

Example Reductions

val xs = Cons(1,Cons(2,Cons(3,Cons(4,Cons(5,Cons(6,Empty)))))



Combining Map and Reduce



Combining Map and Reduce

reduce(0, _+_, map(x => x*x + 1, xs))

Summation

def summation(xs: List, f: Int => Int) = reduce(0, _+_, map(f, xs))

Summation

def square(x:Int) = x * x

summation(xs, square(_)+1)

More Syntactic Sugar

- Functions defined with def can be passed as arguments whenever an expression of a compatible function type is expected
- What constitutes a compatible function type?

Partially Applied Functions

 If we want to pass a function as an argument, but supply some of the arguments to the function ourselves, we can wrap an application to the function in a function literal:

$$map(x => x + 1, xs)$$

Partially Applied Functions

 If we want to pass a function as an argument, but supply some of the arguments to the function ourselves, we can wrap an application to the function in a function literal:

which is equivalent to

Partially Applied Functions

• Eta Expansion: Wrapping a function in function literal that takes all of the arguments of f and immediately calls f with those arguments

(x:Int) => square(x)

is equivalent to

square

Mapping a Computation Over a List

We can use eta expansion to pass operators as arguments:

$$map(x => -x, xs)$$

Mapping a Computation Over a List

We are also using eta expansion when using underscore notation:

map(-_, xs)