Comp 311
Functional Programming

Eric Allen, PhD
Vice President, Engineering
Two Sigma Investments, LLC

Clarification on Homework Assignments

- Really, there are no extensions
    - The real world is no different than this
- We are not providing you with any substantial tests
    - The real world is no different than this
- The specification in the homework assignment is not debatable
    - Constructive feedback for future iterations of the class is always appreciated

Clarification on Homework Assignments

- Ambiguous sections of the homework are open to your interpretation
  - Make a reasonable interpretation and document it
    - You are not bound by subsequent conversations on Piazza that clarify ambiguities
    - We will make every attempt to clarify ambiguities in a reasonable way
- Take instructions concerning file and package names seriously

Lexical vs Dynamic Scoping

- The semantics of function application that we have outlined is referred to as *lexical scoping*

- Early versions of Lisp avoided the need for closures:
  - They reduced function applications by extending the environment in which the application *occurred*
    - This semantics of function application is known as dynamic scoping
    - Why is dynamic scoping problematic?

Additional Syntactic Forms

Repeated Parameters

- 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

```
def squares(xs: Int*) =
  for (x <- xs)
    yield x*x
```

Repeated Parameters

- 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)
```

Repeated Parameters

- 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
```

Repeated Parameters

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

```
squares(1,2,3,4,5) ↦
ArrayBuffer(1, 4, 9, 16, 25)
```

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

Repeated Parameters

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

```
val myArray = Array(1,2,3)
squares(myArray: _*)
```

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

```
sum(1,2,3,4,5)
```

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:

```
Circle()
```

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.

Call-By-Value
and
Call-By-Name

Call-By-Value

- Thus far, the evaluation semantics we have studied (both with the substitution and environment models) is known as call-by-value:

- To evaluate a function application, we first evaluate the arguments and then evaluate the function body

Call-By-Value

- We have seen several “special forms” where this evaluation semantics is not what we want:

`&&` `||` `if-else`

Call-By-Value

- We could delay evaluation in these cases by wrapping arguments in function literals that take no parameters

```
def myOr(left: Boolean, right: () => Boolean) =
  if (left) true
  else right()
```

Call-By-Value

- We could delay evaluation in these cases by wrapping arguments in function literals that take no parameters

```
myOr(true, () => 1/0 == 2) ↦ true
```

- Functions that take no arguments are referred to as *thunks*

Call-By-Name

- Scala provides a way that we can pass arguments as thunks without having to wrap them explicitly

```
def myOr(left: Boolean, right: => Boolean) =
  if (left) true
  else right()
```

We simply leave off the parentheses in the parameter's type

Call-By-Name

- Now we can call our function without wrapping the second argument in an explicit thunk:

```
myOr(true, 1/0 == 2) ↦ true
```

- The thunk is applied (to nothing) the first time that the argument is evaluated in a function

Call-By-Name

- We can use by-name parameters to define new *control abstractions:*

```
def myAssert(predicate: => Boolean) =
  if (assertionsEnabled && !predicate)
    throw new AssertionError
```

Syntactic Sugar: Braces for Passing Arguments

- Any function that takes a single argument can be applied by passing the argument enclosed in braces instead of parentheses

```
myAssert {
  2 + 2 == 4
}
```

Syntactic Sugar: Braces for Passing Arguments

- Any function that takes a single argument can be applied by passing the argument enclosed in braces instead of parentheses

```
myAssert {
  def double(n: Int) = 2 * n
  double(2) == 4
}
```

Sequences of Cases

- Another way to write a function literal is to immediately place a sequence of case clauses in braces:

```
{
  case Some(x) => x
  case None => 0
}
```

Sequences of Cases

```
{
  case Some(x) => x
  case None => 0
}
```

is equivalent to

```
_ match {
  case Some(x) => x
  case None => 0
}
```

Scala Immutable Collections

Immutable Lists

- Behave much like the lists we have defined in class
- Lists are covariant
- The empty list is written `Nil`
- Nil extends `List[Nothing]`

Immutable Lists

- The list constructor takes a variable number of arguments:

```
List(1,2,3,4,5,6)
```

Immutable Lists

- Non-empty lists are built from Nil and Cons (written as the right-associative operator `::`)

```
1 :: 2 :: 3 :: 4 :: Nil
```

List Operations

- `head` returns the first element
- `tail` returns a list of elements but the first
- `isEmpty` returns true if the list is empty
- Many of the methods we have defined are available on the built-in lists

FoldLeft and FoldRight Are Written as Operators

- foldLeft:

```
zero /: xs (op)
```

- foldRight:

```
zero :\ xs (op)
```

SortWith

```
List(1,2,3,4,5,6) sortWith (_ < _)
```

Range

```
List.range(1,5)
```

Using Fill for Uniform Lists

```
List.fill(10)(0) ↦
List(0,0,0,0,0,0,0,0,0,0)
```

Using Fill for Uniform Lists

```
List.fill(3,3)(0) ↦
```

```
List(List(0,0,0),
     List(0,0,0),
     List(0,0,0))
```

Tabulating Lists

```
List.tabulate(3,3) (
  (m,n) => if (m == n) 1 else 0)
)
↦
List(List(1,0,0),
     List(0,1,0),
     List(0,0,1))
```

Immutable Sets

Immutable Sets

- Sets are unordered, unrepeated collections of elements
- Sets are parametric and covariant in their element type

Immutable Sets

```
Set(1,2,3,4,5)
```

Immutable Sets

```
Set(1,2,3) + 4 ↦
Set(1,2,3,4)
```

Immutable Sets

```
Set(1,2,3) - 2 ↦
Set(1,3)
```

Immutable Sets

```
Set(1,2,3) - 4 ↦
Set(1,2,3)
```

Immutable Sets

```
Set(1,2,3) ++ Set(2,4,5) ↦
Set(1,2,3,4,5)
```

Immutable Sets

```
Set(1,2,3) – Set(2,4,5,3) ↦
Set(1)
```

Immutable Sets

```
Set(1,2,3) & Set(2,4,5,3) ↦
Set(2,3)
```

Immutable Sets

```
Set(1,2,3).size ↦
```

3

Immutable Sets

```
Set(1,2,3).contains(2) ↦
true
```

Immutable Maps

Immutable Maps

- Maps are collections of key/value pairs
- They are parametric in both the key and value type
  - Invariant in their key type
  - Covariant in their value type

The -> Operator

- The infix operator `->` returns a pair of its arguments:

```
1 -> 2
```

↦

```
(1,2)
```

The -> Operator is Left Associative

```
> 1 -> 2 -> 3 -> 4
res8: (((Int, Int), Int), Int) = (((1,2),3),4)
```

The Map Constructor

```
Map(“a” -> 1, “b” -> 2, “c” -> 3)
```

↦

```
Map(a -> 1, b -> 2, c -> 3)
```

Map Addition

```
Map(“a” -> 1, “b” -> 2, “c” -> 3) + (“d” -> 4)
```

↦

```
Map(a -> 1, b -> 2, c -> 3, d -> 4)
```

Map Operations

- The operators `-`, `++`, `—`, `map.size` are defined in the expected way

Map Addition

```
Map(“a” -> 1, “b” -> 2, “c” -> 3).contains(“b”)
```

↦

```
true
```

Map Addition

```
Map(“a” -> 1, “b” -> 2, “c” -> 3)(“c”)
```

↦

3

Map Addition

```
Map(“a” -> 1, “b” -> 2, “c” -> 3).keys
```

↦

```
Set(a, b, c)
```

Map Addition

```
Map(“a” -> 1, “b” -> 2, “c” -> 3).values
```

↦

```
Set(1,2,3)
```

Map Addition

```
Map(“a” -> 1, “b” -> 2, “c” -> 3).isEmpty
```

↦

```
false
```

Traits

Traits

- Traits provide a way to factor out common behavior among multiple classes and mix it in where appropriate

Trait Definitions

- Syntactically, a trait definition looks like a class definition but with the keyword “trait”

```
trait Echo {
  def echo(message: String) =
    message
}
```

Trait Definitions

- Traits can declare fields and full method definitions
- They must not include constructors

```
trait Echo {
  val language = “Portuguese"
  def echo(message: String) =
    message
}
```

Using Traits

- Classes “mix in” traits using either the `extends` or `with` keywords

```
class Parrot extends Echo {
  def fly() = {
    // forget to fly and talk instead
    echo(“poly wants a cracker”)
  }
}
```

Using Traits

- Classes “mix in” traits using either the `extends` or `with` keywords

```
class Parrot extends Bird with Echo {
  def fly() = {
    // forget to fly and talk instead
    echo(“poly wants a cracker”)
  }
}
```

Using Traits

- Classes “mix in” traits using either the **`extends`** or **`with`** keywords

```
trait Smart {
  def somethingClever() =
    “better a witty fool than a foolish wit”
}
```

Using Traits

- Classes can mix in multiple traits using either the `with` keywords

```
class Parrot extends Bird with Echo
with Smart {
  def fly() = {
    // forget to fly and talk instead
    echo(somethingClever())
  }
}
```