Comp 311
Functional Programming

Nick Vrvilo, Two Sigma Investments
Robert “Corky” Cartwright, Rice University

September 26, 2017

Announcements

- Homework 1 is due **Thursday**

- Nate has office hours tomorrow
  (check Piazza for details)

- Updated calendar and slides on course website

More on First-Class Functions

More Syntactic Sugar for First-class Functions

- 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:

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

which is equivalent to

```
map(_ + 1, xs)
```

Eta Expansion

η-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 η-expansion to pass operators as arguments:

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

Mapping a Computation Over a List

Note that we are also using η-expansion when we use underscore notation for function literals:

```
map(-_, xs)
```

Returning Functions as Values

We Can Define Functions That Return Other Functions as Values

```
def adder(x: Int): Int => Int = {
  def addX(y: Int) = x + y
  addX
}
```

We Can Define Functions That Return Other Functions as Values

```
def adder(x: Int): Int => Int = {
  def addX(y: Int) = x + y
  addX
}
```

The explicit return type is needed because Scala type inference assumes an unapplied function is an error

We Can Define Functions That Return Other Functions as Values

```
def adder(x: Int) = {
  def addX(y: Int) = x + y
  addX _
}
```

Alternatively, we can η-expand `addX` to assure the type checker that we really do intend to return a function

We Can Define Functions That Return Other Functions as Values

```
def adder(x: Int) = {
  def addX(y: Int) = x + y
  addX _
}
```

An underscore outside of parentheses in a function application denotes the entire tuple of arguments passed to the function is left unapplied

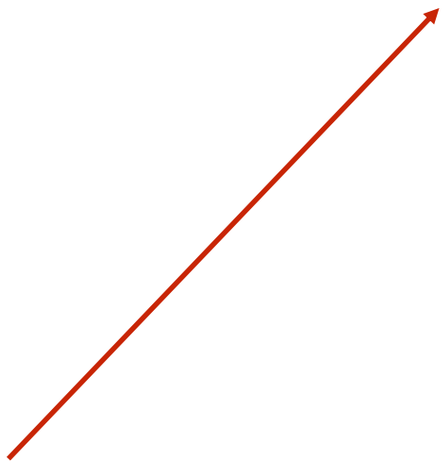
We Can Define Functions That Return Other Functions as Values

```
def adder(x: Int) = x + (_: Int)
```

We can instead define add by *partially* η-expanding the + operator. But then we need to annotate the second operand with a type.

We Can Define Functions That Return Other Functions as Values

```
def adder(x: Int): Int => Int = x + _
```



If we have the explicit return type, then the compiler has all the information it needs to correctly infer the type

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
```

Limiting Visibility

Visibility Modifier Private

For a method `Arithmetic.reduce` in package `Rationals`

Modifier	Explanation
no modifier	public access
`private`	private to object Arithmetic

Local Definitions

- As with constant definitions (`val`), we can make function definitions local to the body of a function

- The functions can be referred to only in the body of the enclosing function

Local Definitions

```
def reduce() = {
  val isPositive =
    ((numerator < 0) & (denominator < 0)) |
      ((numerator > 0) & (denominator > 0))

  def reduceFromInts(num: Int, denom: Int) = {
    require ((num >= 0) & (denom > 0))
    val gcd = Arithmetic.gcd(num, denom)
    val newNum = num/gcd
    val newDenom = denom/gcd

    if (isPositive) Rational(newNum, newDenom)
    else Rational(-newNum, newDenom)
  }
  reduceFromInts(Arithmetic.abs(numerator), Arithmetic.abs(denominator))

} ensuring (_ match {
  case Rational(n,d) => Arithmetic.gcd(n,d) == 1 & (d > 0)
})
```

Local Imports

Unlike Java, Scala's import statements are *not* limited to the top-level. They can appear almost anywhere:

```
def myHelperMethod(...) = {
  import Arithmetic._
  val someVal = gcd(abs(x), abs(y))
  // ...
}
```

Additional Syntactic Forms

Repeated Parameters

- Scala allows the last entry in a parameter list to stand for zero or more arguments

- Arguments are placed in a sequence of the given type

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

Repeated Parameters

- Scala allows the last entry in a parameter list to stand for zero or more arguments

- Arguments are placed in a sequence 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 entry in a parameter list to stand for zero or more arguments

- Arguments are placed in a sequence of the given type

```
def fnName(a1: T1, ..., aN: TN*) = expr
```

Repeated Parameters

- Scala allows the last entry in a parameter list to stand for zero or more arguments

- Arguments are placed in a sequence of the given type

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

ArrayBuffers

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

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(distance = 2.0, time = 5.0)
```

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 it's unclear which arguments correspond to which parameters, e.g.:
  - There are multiple arguments of the same type
  - 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.

Takeaway Points

- Choose the syntactic construct that makes your first-class functions clear and concise.

- Scala's import statements are flexible. Try to cut the verbosity without introducing ambiguity.

- Scala gives you several tools to limit visibility / access (This is important! Think *encapsulation.*)

- Syntactic sugar can help or hurt—think before using.