

Comp 311

Functional Programming

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

September 26, 2019

Announcements

- Homework 1 is due **Tuesday**
- Our new TA has office hours on *Monday*
(check Piazza announcement for details)
- Homework 2 will also be posted on *Tuesday*

Guidelines On Using Function Literals

- Function literals are well-suited to situations in which:
 - The function is only used once
 - The function is not recursive
 - The function does not constitute a key concept in the problem domain

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(y+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): 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) ↪*
```

```
Cons(-1, Cons(-2, Cons(-3, Cons(-4, Cons(-5, Cons(-6, Empty))))))
```

```
map(x => x+x, xs) ↪*
```

```
Cons(1, Cons(4, Cons(6, Cons(8, Cons(10, Cons(12, 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

$$\sum_{x \in X} x$$

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

$$\prod_{x \in X} x$$

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) ↦* 21
```

```
reduce(1, (x,y) => x * y, xs) ↦* 720
```


Min and Max

```
def max(xs: List): Int =  
  reduce(Int.MinValue, (x,y) => if (x > y) x else y, xs)  
  
def min(xs: List): Int =  
  reduce(Int.MaxValue, (x,y) => if (x < y) x else y, xs)
```

Min and Max

Numbers in Scala have min/max binary operators:

```
def max(xs: List): Int =  
  reduce(Int.MinValue, (x,y) => x max y, xs)
```

```
def min(xs: List): Int =  
  reduce(Int.MaxValue, (x,y) => x min y, xs)
```

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,

```
((x: Int, y: Int) => (x + y))
```

becomes

```
_ + _
```

Example Reductions

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

```
reduce(0, _+_, xs) ↪* 21
```

```
reduce(1, _*__, xs) ↪* 720
```



Note the multiple parameters

Min and Max, Simplified

```
def max(xs: List) = reduce(Int.MinValue, _ max _, xs)
def min(xs: List) = reduce(Int.MaxValue, _ min _, xs)
```

Combinations of Maps and Reductions

$$\sum_{x \in \mathcal{X}_S} x^2 + 1$$

Combinations of Maps and Reductions

```
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 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 Reduction

η -expansion: Wrapping a function in function literal that takes all of the arguments of f and immediately calls f with those arguments

η -reduction: Reducing a function literal that simply forwards all of its arguments with the target function

`(x: Int) => square(x)`

can be η -reduced 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 )
```

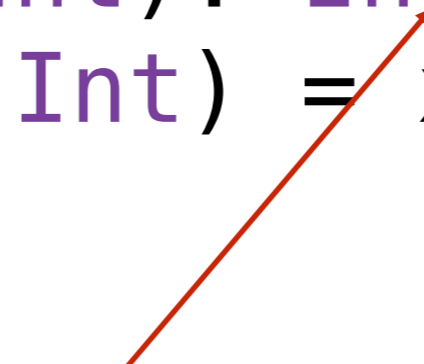

Returning Functions as Values

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

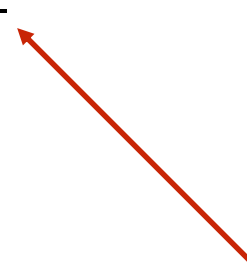
```
def adder(x: Int): Int => Int = {  
  def addX(y: Int) = x + y  
  addX  
}
```



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

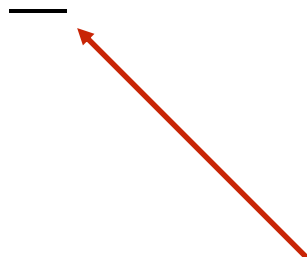
```
def adder(x: Int) = {  
  def addX(y: Int) = x + y  
  addX  
}
```



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

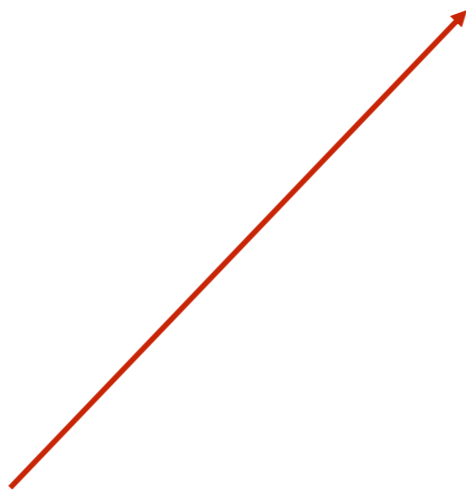
We Can Define Functions That Return Other Functions as Values

```
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
<i>no modifier</i>	public access
<code>private</code>	private to object <code>Arithmetic</code>

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))  
  // ...  
}
```

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.