Announcements

- Quiz 8
- Rainbow grades: HW1-8, Quiz1-6, Exam1-2
- Still grading: HW9, Quiz 7
- HW10 due today
- HW11 out today, due Friday

Today’s Lecture Outline

- Scala: a functional and object-oriented programming language
- Key ideas
  - Lists: cons, map/reduce, immutable types
  - Scala is functional: anonymous functions, higher-order functions, call-by-name parameters
  - Scala is object-oriented: classes, singleton objects, case classes and pattern matching, traits and mixins
  - And much more…

Scala Resources

- [http://www.scala-lang.org](http://www.scala-lang.org)
  - Official Scala website
  - Scala API
- [https://docs.scala-lang.org/tutorials/scala-for-java-programmers.html](https://docs.scala-lang.org/tutorials/scala-for-java-programmers.html)
  - Tutorial
- Getting started: tutorial + slides

Scala

- General-purpose language
- Multi-paradigm: focus is on functional and object-oriented programming
- Designed by Martin Odersky at EPFL, first version around 2001
- Built on the Java Virtual Machine (JVM)
- Used at Twitter, LinkedIn, Spark big data

Model for variables?
- Reference model
- Scoping discipline?
  - Static scoping
- Typing discipline?
  - Static typing, Type annotations, some type inference
  - Subtype, and (explicit) parametric polymorphism
- Parameter passing?
  - Mixed modes. Default by-value, allows by-name
Getting Started

- Download Scala
  - On Mac: brew install scala
- Scala REPL:
  - `$ scala`
  - `$ scala`
- Compiling and running Scala
  - `$ scalac Expressions.scala`
  - `$ scala Expressions`

Defining Variables

- `var` can be reassigned
  - `$ var x = 1`
  - `x: Int = 1`
  - `$ x = 42`
  - `x: Int = 42`
- `val` cannot be reassigned
  - `$ val y = 1`
  - `y: Int = 1`
  - `$ y = 42`

Lists

- Lists are important in Scala too!
  - `$ var li = List(1,2,3)`
  - `li: List[Int] = List(1, 2, 3)`
- We can have elements of different types
  - `$ var li2 = 1::2::3::Nil`
  - `li2: List[Int] = List(1, 2, 3)`
- `Any` is the supertype of all Scala types

Basic Control Structures

- If-then-else
  - `$ if (n%2==0) print("even") else print("odd")`
  - `n%2==0 { print("even") } else { print("odd") }`
- While
  - `$ var x = 0`
  - `$ while (x < 10) { println(x); x = x+1; }`
- In Scala, everything is an expression!
  - `$ var a = while (x < 10) { println(x); x = x+1; }`
  - `a: Unit = ()`  // essentially Java’s void

Lists

- Cons and append
  - `$ val li1 = List(1,2)`
  - `$ val li2 = List(3,4)`
  - `$ li2::li3`  // yields what?
  - `res0: List[Any] = List(List(1, 2), 3, 4)`
  - `$ val li2::::li3`  // is append!
  - `res1: List[Int] = List(1, 2, 3, 4)`

Lists

- `map`, `foldLeft`, `foldRight`, `filter` are built in
  - `def foldLeft[B](z: B)(op: (B, A) ⇒ B): B`
  - `foldLeft` is a function applied on receiver List[A]. It takes z of type B, and a binary operation op; it returns result of type B.
  - `$ li1::foldLeft(0)(_+_)`  // same semantics as our `foldl`
  - `res6: Int = 10`
  - `$ li1.foldRight(0)(_-_)`  // same semantics as our `foldr`
  - `res7: Int = -2`
Lists

- Lists are immutable!
  - Scala supports immutable types (Lists, Strings) as a defense against call-by-sharing
  - Just as in Scheme, :: and ::: create new lists

- Arrays are mutable
  scala> li1(3) = 5
  error ...
  scala> val a = Array(1,2,3,4)
  a: Array[Int] = Array(1, 2, 3, 4)
  scala> a(3) = 5

Lists (sequence) comprehensions

- In Scala, known as for comprehensions
- for (seq) yield expression
- seq includes generators, may include filters
  scala> def all_pairs(lis1:List[Int], lis2:List[Int]) = for (x<-lis1; y<-lis2) yield ((x,y))
  all_pairs: (lis1: List[Int], lis2: List[Int])List[(Int, Int)]
  scala> all_pairs(List(1,2,3),List(6,5,4)) // yields what?
  res9: List[(Int, Int)] = List((1,6), (1,5), (1,4), (2,6), (2,5), (2,4), (3,6), (3,5), (3,4))

Lists and Streams

- Scala streams are like lists, however, elements are computed lazily
- Can create infinite lists such as [1..] in Haskell
  scala> def fun (n:Int) : Stream[Int] = n #: fun(n+1)
  fun: (n: Int)Stream[Int]
  scala> fun(1).take(10) // yields what?
  scala> fun(1)(100) // yields what?
  res3: Int = 101
  Compare to def fun2 (n:Int) : List[Int] = n :: fun2(n+1)
  fun2(1)

Functions

- Functions that are part of an object are called methods (more on classes, case classes and singleton objects later)

  object HelloWorld {
    def main(args: Array[String]) {
      println("Hello, world!")
    }
  }

- In Scala, functions, just like any other value, are objects

Lecture Outline

- Scala: a functional and object-oriented programming language
- Key ideas
  - Lists: cons, map/reduce, immutable types
  - Scala is functional: anonymous functions, higher-order functions, call-by-name parameters
  - Scala is object-oriented: classes, singleton objects, case classes and pattern matching, traits and mixins

Nested Functions and Scoping

var i = 1; var j = 3;
def outer() : (Int,Int,Int) = {
  var i = 2;
  def middle(k:Int) : (Int,Int,Int) = {
    def inner() {
      var i = 4; println(i); }
    inner();
    (i,j,k) // middle returns tuple
  }
  middle() // outer returns tuple
}
def outer(); println (i++,"+"j);
Defining Functions

```
scala> def square(x: Int) = x*x
square: (x: Int)Int // Scala infers return type

Or
scala> def square(x: Int) = { x*x }

square: (x: Int)Int // Also, def square(x: Int) : Int = x*x
```

res0: Int = 25

But not
```
scala> def square(x: Int) : Int = x*x
```

res10: Int = 10

Higher-order Functions

Of course, higher-order functions everywhere

```
scala> def plusone(x: Int) = x+1

plusone: (x: Int)Int
```

Note that Scala infers return type
```
scala> def apply_n[A](f: A => A, n: Int, x: A) : A = 
  if (n==0) x else apply_n(f, n-1, f(x))

apply_n: [A](f: A => A, n: Int, x: A)A
```

```
scala> apply_n(plusone, 10, 0)
res10: Int = 10
```

Anonymous Functions

Functions are first-class values (objects)

- Can be passed as arguments, returned, assigned
```
scala> var lis = List(1,2,3,4)
scala> lis.map((x: Int) => x+1) // yields what?
res3: List[Int] = List(2, 3, 4, 5)
scala> val plusone = (x: Int) => x+1
```

Different from def:
```
scala> def plusone(x: Int) = x+1
```

Type Inference

Scala does only simple type inference...

- No rule to say when it can infer, and when it can’t
```
scala> val plusone = x => x+1
<console>:11: error: missing parameter type
```

- Have to type parameter
```
scala> val plusone = (x: Int) => x+1
```

- Scala can infer type if anonymous function immediately applied

Call-by-name Parameters

By default Scala uses call-by-value
```
scala> def square(x: Int) = x*x
scala> square(1+2)
1+2 = 3
3*3 = 9
```

- Can specify call-by-name by using =>
```
scala> def square(x: => Int) = x*x
scala> square(1+2)
(1+2)*(1+2) = 3*(1+2)
3*3
9
```

HW10 Example (Attempt 1)

```
def compute(first : Int, last : Int, incr : Int, 
i : =>Int, term : =>Double) = {
  var result : Double = 0.0;
  i = first;
  while (i <= last) {
    result = result + term;
    i = i + incr;
  }
  result
}
```

```
var i = 0; var a = Array(1,2,3); val s = compute(0,2,1,i(a(i)));
```
HW10 Example (Attempt 2)

```scala
def compute(first:Int, last:Int, incr:Int, j:Int => Unit, term: => Double) = {
  var result : Double = 0.0;
  var i : Int = first;
  while (i <= last) {
    result = result + term; // term evaluated in ref. env. of caller
    j(incr); // increments main's i
    i = i+incr;
  }
  result
}
var i : Int = 0;
var a = Array(1,2,3);
def inc(incr:Int) { i = i+incr; }
print(compute(0,2,1,inc _,a(i)));
```

Objects

- In Scala, everything is an object
  - Including functions and numbers!
  - Unlike Java, no distinction between primitive types (value model), and reference types
- Numbers are objects
  - `scala> 1+2*3`
  - Or, we can use "instance call" syntax
    - `scala> (1).+(2).*(3) // yields what?`
    - `scala> (2::Nil).:+(1) // yields what?`

- Functions are objects too
  - Means they are “first-class” values, can be passed, returned and assigned
  - Anonymous functions, curried functions, higher-order functions, etc.
  - Essential characteristic of Scala, renders Scala a “functional” language

Objects

- Singleton objects have a single instance. In Java, we needed the Singleton pattern
  ```scala
  object HelloWorld {
    def main(args: Array[String]) {
      println("Hello, world!")
    }
  }
  ```
  - Object declaration entails a singleton object
    - Above declares both a class and an object
    - No static methods/fields in Scala!
Case Classes

- Class `Tree` represents ASTs of arithmetic expressions (e.g., `x+x+(1+y)`)
  
  ```scala
  abstract class Tree
  case class Sum(l: Tree, r: Tree) extends Tree
  case class Var(n: String) extends Tree
  case class Const(v: Int) extends Tree
  ```

- Can be viewed as a classical OO hierarchy (subtype polymorphism, dynamic dispatch)
- Can be viewed as an Algebraic Data Type (as in Haskell, a tagged union of products)

Pattern Matching

- We already know what this is!
  
  ```scala
  def eval(t: Tree, env: Environment): Int = t match {
    case Sum(l, r)  => eval(l, env) + eval(r, env)
    case Var(n)      => env(n)
    case Const(v)  => v
  }
  ```

- Just as in Haskell
  - Test: does value `t` match pattern (e.g., `Sum`)
  - Binding: if yes, then bind components to corresponding variables (e.g., `l` an `r`)

HW11

- Traversal of composite expressions
  - A-a-again...
  - This time in Scala

- One problem: traversal using pattern matching over a hierarchy of case classes
- Another problem: Visitor-like traversals using a higher-order method

Traits

- A class can mix in code from one or more traits
  
  ```scala
  trait Ord {
    def < (that: Any): Boolean
    def <=(that: Any): Boolean = (this < that) || (this == that)
    def > (that: Any): Boolean = !(this <= that)
    def >=(that: Any): Boolean = !(this < that)
  }
  ```

- In Scala, `==` goes to `equals`, i.e., value equality!
A class can mix in code from one or more traits

```scala
class Date(y: Int, m: Int, d: Int) extends Ord {
  def year = y
  def month = m
  def day = d
  override def toString(): String = year + "." + month + "." + day
}
```

In Scala, `override` annotation is mandatory when overriding a concrete method.

### Traits and Mixins

- An abstract class can extend an abstract class
- There may be conflicting trait members

```scala
abstract class AbsIterator {
  type T
  def hasNext: Boolean
  def next(): T
}

class StringIterator(s: String) extends AbsIterator {
  type T = Char
  private var i = 0
  def hasNext = i < s.length // must implement member
  def next() = {
    val ch = s.charAt i
    i += 1
    ch
  }
}
```

- Can mix-in functionality from different traits
- As in Java, one `extends` multiple `with` (implements)
- More flexible (and more complex) than Java
- Trait can extend an abstract class
- There may be conflicting trait members

```scala
trait RichIterator extends AbsIterator {
  def foreach(f: T => Unit): Unit = while (hasNext) f(next())
}

class RichStringIterator extends StringIterator with RichIterator {
...}
```