Goals

• Understand PL features in a general way
  – Be able to understand and apply concepts
    • Makes you a better programmer in any language

• Expand your horizons
  – Learn two new languages
  – Learn to think outside the C/C++/Java box. You are limited by the languages you know.

• Learn new PLs quickly
Theory vs Practice

• 85% Practice
  – Looking at languages with the view of writing programs that do useful things.
  – How does feature X help me write a better program (for some definition of “better”).

• 15% Theory
  – Puts PL features into perspective
  – Formal syntax. Turing Machines. Lambda calculus.
Focus Language: Scala

• To make the concepts concrete we will focus on one particular language
  – Scala (http://www.scala-lang.org)
    • Functional/OO hybrid language
    • Targets the JVM. Easily mixes with Java.
    • Very rich with many interesting features from a PL theory perspective.
    • Very practical. Large community. Relatively good tool support. Could become an important language in the near future.
Focus Language (continued)

• No language can illustrate all concepts
  – Scala is not...
    • A dynamic language
    • A logic language
    • A systems or embedded language [some might disagree]

• We will talk about these things as well
  – This course is not “all Scala all the time.”
    • ... it is “mostly Scala most of the time.”
Scala Targets JVM

• **Good**
  – Can access huge collection of Java libraries
  – Can take advantage of advanced Java technology
  – Can be deployed anywhere Java can

• **Bad**
  – Tied to the Java ecosystem
  – Suffers disadvantages of any VM based language
Language Categories

• **Imperative** (also Object Oriented)
  – Program a sequence of commands (imperatives)
  – Each command modifies the state of memory

• **Functional**
  – Program is a large expression that is evaluated
  – All data is immutable (no state modified or side effects created during evaluation)

• **Logic**
  – Program is a set of rules that describe the solution
  – System finds a result that obeys all the rules
Scala is Imperative

```scala
def sieve(max: Int): Array[Boolean] = {

  // Create and initialize the array.
  val flags = new Array[Boolean](max)
  for (i <- 0 until max) flags(i) = true

  // Zero and one are not prime.
  flags(0) = false
  flags(1) = false

  // Sieve off the non-primes.
  for (i <- 2 until max) {
    if (flags(i) == true) {
      for (j <- 2*i until max by i) flags(j) = false
    }
  }

  // Return the result.
  flags
}
```
Scala is Object Oriented

// Abstract superclass describes all animals.
abstract class Animal {
  def weight: Double
}

// Subclass representing cats. Overrides abstract methods.
class Cat(w: Double) extends Animal {
  if (w < 0.0) throw new BadWeightException

  def weight = w
}

// Method to compute total weight of all animals in a list.
def totalWeight(zoo: List[Animal]) =
  zoo.map(_.weight).foldLeft(0.0)(_ + _)

// Send a list of Cats to the totalWeight method.
val catFarm = List(new Cat(8.5), new Cat(5.2), new Cat(523.0))
val catWeight = totalWeight(catFarm)
Scala is Functional

// Return the total size of all files in the specified folder.
def folderSize(folderName: String) = {

    // Java libraries are usable from Scala.
    val folder = new java.io.File(folderName)

    // Process list of files using “higher order” methods.
    val fileLengths =
        folder.listFiles filter { _.isFile } map { _.length }

    // Collapse the resulting array of file lengths into a single value.
    fileLengths.foldLeft(0L)(_ + _)
}
Scala Integrates OO and FP

// Class extends the type “function taking String returning Int”
class NameConverter extends String => Int {
    // Method to use when instance is “called” as a function.
    def apply(s: String) = { ... }

    // Some other method.
    def configure(base: Int) = { ... }
}

val converter = new NameConverter
converter.configure(16)  // It’s an object!
val result = converter("Peter")  // It’s a function!

// Method taking a function of type String => Int as a parameter.
def workWith(operation: String => Int) = { ... }

// Can pass a NameConverter; it’s a subtype of String => Int
workWith(converter)
Domain Specific Languages

• A language designed for use in a specific application domain (by “domain experts”)
  – Gnuplot
  – PIC
  – MATLAB/Octave
  – LabView
  – TeX
  – Macro languages of various kinds
  – Many others...
External vs Internal DSLs

• External
  – DSL creator writes a program that processes the new language
  – DSL processor can be in any language
  – DSL processor uses compiler techniques
  – Example: Gnuplot is written in C

• Internal
  – DSL creator extends a “host” language to add new syntax for the DSL
  – DSL user can drop to the host language at any time
Scala and DSLs

• Scala has features to support internal DSLs
  – Flexible syntax. You can (with limitations) add:
    • New keywords
    • New operators
    • New control structures

• Enables “DSL oriented programming”
  – Don’t write a program to solve your problem...
  – Create a DSL that makes the problem easy
    • ... and then easily solve it with your DSL
import org.scalatest.FlatSpec
import org.scalatest.matchers.ShouldMatchers

class StackSpec extends FlatSpec with ShouldMatchers {
    "A Stack" should "pop values in last-in-first-out order" in {
        val stack = new Stack[Int]
        stack.push(1)
        stack.push(2)
        stack.pop() should equal (2)
        stack.pop() should equal (1)
    }

    it should "throw NoSuchElementException if an empty stack is popped" in {
        val emptyStack = new Stack[String]
        evaluating {
            emptyStack.pop()
        } should produce [NoSuchElementException]
    }
}

From: http://www.scalatest.org
Example DSL: Parser Combinators

This is Scala

```scala
def inclusion_credential: Parser[RTInclusionCredential] =
  role_definition ~ "<-" ~ role_definition ^^
  { case target ~ "<-" ~ source =>
    RTInclusionCredential(target, source) }

def role_definition: Parser[(String, String)] =
  entity ~ "." ~ role_identifier ^^
  { case entityName ~ "." ~ roleName =>
    (entityName, roleName) }

def entity: Parser[String] =
  ident

def role_identifier: Parser[String] =
  ident

Can parse strings like “A.r <- B.s”
```
Example DSL: Telnet State Machine

This is Scala

override val transitions: Seq[Transition] =
  (data, IAC) -> cmd ::
  (data, 0)   -> data ::
  (data, 10)  -> data ::
  (data, 13)  -> data + eatLine + echo (""") ::
  (data, {_:Event=>true}) -> data + eatChar + echo (""") ::
  (cmd, IAC)  -> data ::
  (cmd, Seq(WILL, WONT, DO, DONT)) -> neg + push ::
  (neg, {_:Event=>last==SM(DO)})  -> data + mode(true) + pop ::
  (neg, {_:Event=>last==SM(DONT)}) -> data + mode(false) + pop ::
  (neg, AnyEvent) -> data + echo("interesting sequence...") + pop ::
  (cmd, SB)   -> subneg ::
  (""".*"""".r, CR) -> data ::
  Nil

Posted on the Scala User’s mailing list. See also: http://blog.razie.com/search/label/dsl
Static vs Dynamic

• Static Languages
  – Perform many program checks at compile time (before the program runs)
    • e.g. Static type checking
  – Generally require all code references to be resolved ahead of time
  – Generally do not allow programs to execute data
    • For example, read a string from the user containing program text and then execute that code.
Static vs Dynamic (continued)

• Dynamic Languages
  – Postpone many language checks until run time
    • e.g. Dynamic type checking
  – Can easily load code at run time
  – Often allow the execution of code stored in data objects
Static vs Dynamic (continued)

• Static Languages...
  – ... are fast. Since checks are done by the compiler they need not be done while the program runs
  – ... are robust. Many errors are found by the compiler.
  – ... are less flexible. The program can’t as easily adapt to new conditions once compiled.
  – ... are less interactive. It is difficult to modify the code of the program while it runs.

• Dynamic Languages...
  – ... are the opposite!
Python is Dynamic

This is Python

```python
def sum(x, y): return x + y

z = sum(1, 2)  # Computes 3
z = sum(1.0, 2.0)  # Computes 3.0
z = sum("Hello", "World")  # Computes “HelloWorld”
z = sum("Hello", 2)  # Run time error
```

The last line throws a TypeError exception...

“TypeError: Can’t convert ‘int’ object to str implicitly”
Scala is Static

This is Scala

def sum(x: Int, y: Int) = x + y

z1 = sum(1, 2)                 // Computes 3
z2 = sum(1.0, 2.0)             // Compile time error
z3 = sum(“Hello”, “World”)     // Compile time error
z4 = sum(“Hello”, 2)           // Compile time error

You can use a type class to generalize sum over all numeric types

def sum[A](x: A, y: A)(implicit n: Numeric[A]) = n.plus(x, y)

z1 = sum(1, 2)                 // Computes 3
z2 = sum(1.0, 2.0)             // Computes 3.0
z3 = sum(“Hello”, “World”)     // Compile time error (not numeric)
z4 = sum(“Hello”, 2)           // Compile time error
Class Organization

• Lecture driven by slides and demonstrations
• Handouts, slides, assignments on my web site: http://web.vtc.edu/users/pcc09070/cis-3030
  – See web site for grading policy, late policy, etc
  – Your first assignment has been posted!
• Homework submitted electronically
• Tests take home and/or on Moodle
Semester Project

• Research a programming language of your choice and...
  – Write (at least) two programs using it
  – Do a 15 minute oral presentation in front of class
  – Write a 3-5 page report about your language

• Details on class web site
  – First due date: Choose your language by September 19.
Good Luck!

And don’t forget to have fun!