Comp 311
Functional Programming

Eric Allen, Two Sigma Investments
Robert “Corky” Cartwright, Rice University
Sağnak Taşırlar, Two Sigma Investments

Some Additional Scala Features

Scripting in Scala

- Scala is designed for building large-scale systems
- It also scales down to small scripts:
  - In a single file, we can place class definitions, function definitions, and even top-level expressions

Scripting in Scala

- In a single file `hello.scala`, write:

```
println("Hello, scripting world!")
```

- From the command-line (in an environment where scala has been installed):

```
scala hello.scala
```

Scripting in Scala

- Command-line arguments are available via a global array named `args`:

```
println(“Hello, ” + args(0) + “!”)
```

Scripting in Scala

- At the shell:

```
scala hello.scala Owls
```

- And the result is:

```
Hello, Owls!
```

Scripting in Scala

- On Unix, you can run a Scala script directly from the shell by putting the following at the top of your script (let’s name the file `hello`):

```
#!/bin/sh
exec scala “$0” “$@”
```

- Then make the file executable:

```
chmod u+x hello
```

Scala Applications

- To compile a stand-alone Scala application, you can put the driver into a singleton object with a `main` method

Scala Applications

- Any singleton object might contain a main method that takes an argument of type `Array[String]`:

```scala
package edu.rice.cs.comp311.lectures.lecture22

object ArgLengths {
  def main(args: Array[String]) = {
    for (arg <- args)
      println(arg + ": " + arg.length)
  }
}
```

Scala Applications

- Compile using `scalac` or `fsc`
  - `scalac` will recompile all referenced jars, files,…
    - Therefore, it can be slow
  - `fsc` starts a process the first time it is run that memoizes compilation of referenced files

Scala Applications

- Execute a compiled classfile using the `scala` command
- Include the full path name

```
scala edu.rice.cs.comp311.lectures.lecture22.ArgLengths
```

Fields in Non-Case Classes

- constructor of a class is a function:
  - When it is called, the enclosing environment is extended and an object is returned, as defined by the body of the class

Fields in Non-Case Classes

- A natural consequence:
  - The arguments to a constructor call are not directly accessible outside the object that is returned from the call
- To make a parameter accessible, define a field
- Case classes automatically define a field for every constructor parameter

The Follow Code Will Not Pass Type Checking

```
class Rational(numerator: Int, denominator: Int) {
  def +(that: Rational) =
    new Rational(numerator * that.denominator +
                 that.numerator * denominator,
                 denominator * that.denominator)
}
```

Declaring the Fields Explicitly Fixes The Problem

```
class Rational(n: Int, d: Int) {
  val numerator = n
  val denominator = d

  def +(that: Rational) =
    new Rational(numerator * that.denominator +
                 that.numerator * denominator,
                 denominator * that.denominator)
}
```

Auxiliary Constructors

- Scala allows for multiple constructor declarations
- Additional constructors are defined as methods with name `this`
- The first action of an auxiliary constructor must be to invoke another constructor
    - Only constructors defined earlier in the class definition are in scope

Auxiliary Constructors

```
class Rational(n: Int, d: Int) {
  val numerator = n
  val denominator = d

  def this(n: Int) = this(n, 1)

  def +(that: Rational) =
    new Rational(numerator * that.denominator +
                 that.numerator * denominator,
                 denominator * that.denominator)
}
```

Companion Objects

- A class can be given a *companion object*:
  - A singleton object definition with the same name
  - Must be defined in the same file as the class
  - The object and class share private members

Companion Objects and Factory Methods

- Companion objects are well-suited for defining factory methods:

```
object Rational {
  def apply(n: Int, d: Int) =
    if (d != 0) new Rational(n, d)
    else throw new Error("Given a zero denominator")
}
```

Private Primary Constructors

- Primary constructors can be hidden by prefixing them with the keyword private:

```
class Rational private(n: Int, d: Int) {
  val numerator = n
  val denominator = d

  def this(n: Int) = this(n, 1)

  def +(that: Rational) =
    new Rational(numerator * that.denominator +
                 that.numerator * denominator,
                 denominator * that.denominator)
}
```

Private Constructors and Companion Objects

```
> Rational(1,1)       // ok
> Rational(1,0)       // error
> new Rational(1,2)   // error
> new Rational(2)     // ok
```

Extractors

Extractors

- It is possible to control how an object will interact with pattern matching through the use of *extractors*
- Extractors are objects that define an unapply method, which takes an object and returns an option of one or more elements

Extractors

```scala
object Rational {
  def apply(n: Int, d: Int) = {
    if (d != 0) new Rational(n, d)
    else throw new Error("Given a zero denominator")
  }

  def unapply(q: Rational): Option[(Int, Int)] = {
    Some((q.numerator, q.denominator))
  }
}
```

Extractors

- An unapply method is called in a pattern by prefixing the name of the extractor object followed by a tuple of expected elements
- If the unapply method returns Some((x1,…xN)) and the arity of the tuple (x1,…xN) matches the number of bound variables in the pattern, we have a match

Extractors

```scala
class Rational private(n: Int, d: Int) {
  val numerator = n
  val denominator = d

  def +(that: Rational) = {
    that match {
      case Rational(n2,d2) =>
        Rational(n * d2 + n2 * d,
                 d * d2)
    }
  }
}
```

Case Classes Revisited

- We are now in a position to better explain what a case class definition is given implicitly:
  - A private primary constructor
  - Immutable fields for every parameter
  - Structural `equals` and `hashCode` methods
  - A structural `toString` method
  - A companion object with `apply` and `unapply` methods
  - A `copy` method with parameters for each constructor parameter, defaulted to the field values of the receiver

Extractors vs Case Classes

- Explicit extractors are more verbose than using case classes
- However, they have advantages of their own:
  - separates implementation from pattern matching
  - can deconstruct objects outside of their class definitions
  - can perform more sophisticated deconstruction
    - e.g. regular expression matching on strings

Extractors vs Case Classes

- Case classes also have many advantages:
  - Conciseness
  - Performance: Scala compiler optimizes patterns with case classes aggressively

Combinator Parsing

Combinator Parsing

- Sometimes there are situations in which we need to process expressions in a small ad-hoc language
  - Configuration files for your program
  - An input language to your program such as search queries

Combinator Parsing

- Options:
  - Roll your parser
    - Requires significant expertise and time
  - Use a parser generator (ANTLR)
    - Many advantages but also requires learning and wiring up a new tool into your program

Combinator Parsing

- Another option:
  - Define an *internal domain-specific language*
  - Consists of a library of *parser combinators:*
    - Scala functions and operators that serve as the building blocks for parsers

Combinator Parsing

- Each combinator corresponds to one *production* of a context-free grammar

Arithmetic Expressions

```
  expr ::= term {“+” term | “-” term}.
  term ::= factor {“*” factor | “/” factor}.
factor ::= floatingPointNumber | “(” expr “)”.
```

Arithmetic Expressions

```
  expr ::= term {“+” term | “-” term}.
  term ::= factor {“*” factor | “/” factor}.
factor ::= floatingPointNumber | “(” expr “)”.
```

Denotes definition of a production

Arithmetic Expressions

```
  expr ::= term {“+” term | “-” term}.
  term ::= factor {“*” factor | “/” factor}.
factor ::= floatingPointNumber | “(” expr “)”.
```

Denotes alternatives

Arithmetic Expressions

```
  expr ::= term {“+” term | “-” term}.
  term ::= factor {“*” factor | “/” factor}.
factor ::= floatingPointNumber | “(” expr “)”.
```

Denotes zero or more repetitions

Arithmetic Expressions

```
  expr ::= term {“+” term | “-” term}.
  term ::= factor {“*” factor | “/” factor}.
factor ::= floatingPointNumber | “(” expr “)”.
```

Square brackets [ ] denote optional occurrences (not used here).

Example Arithmetic Expression

```
2 * 3 + 4 * 5 - 6
```

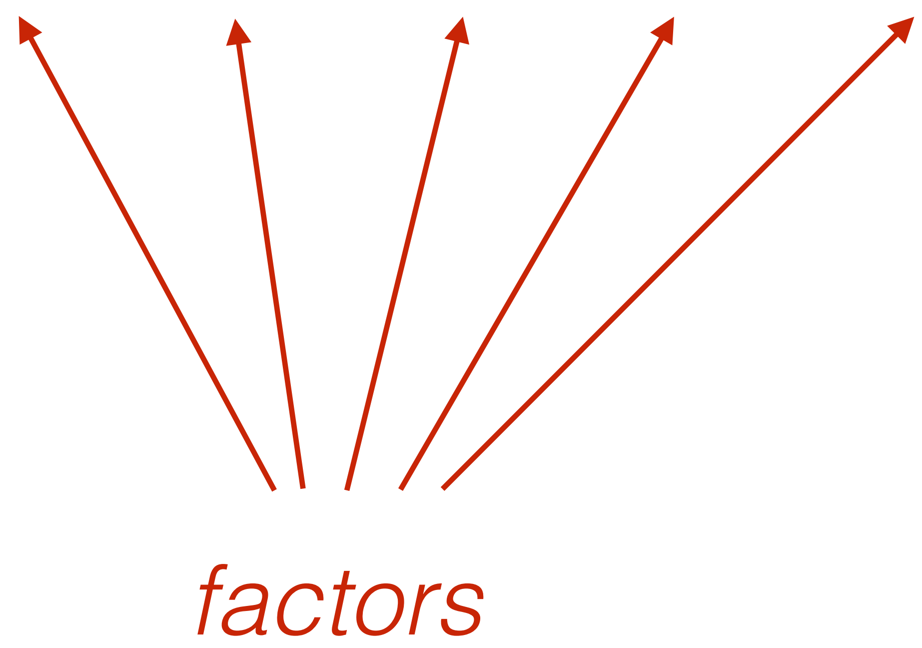
A Formal Grammar for Arithmetic Expressions in BNF

```
  expr ::= term {“+” term | “-” term}.
  term ::= factor {“*” factor | “/” factor}.
factor ::= floatingPointNumber | “(” expr “)”.
```

Denotes one or more repetitions

Example Arithmetic Expression

`2 * 3 + 4 * 5 - 6`

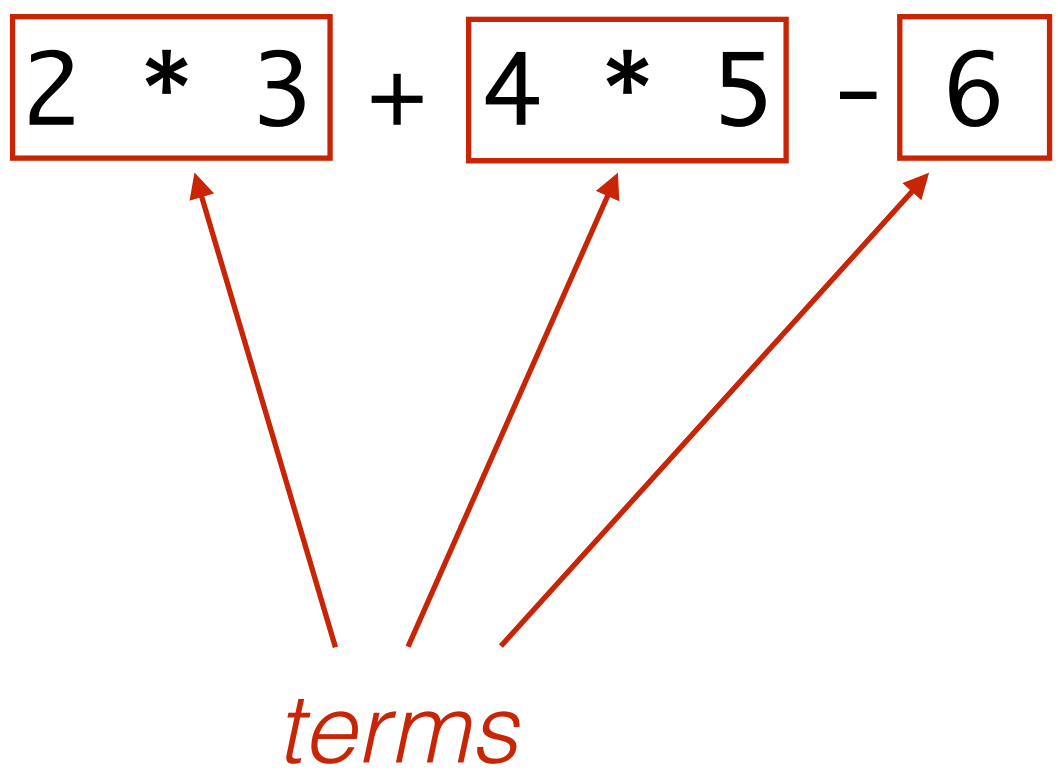


Arithmetic Expressions

```
  expr ::= term {"+" term | "-" term}.
  term ::= factor {"*" factor | "/" factor}.
factor ::= floatingPointNumber | "(" expr ")".
```

Denotes one or more repetitions

Example Arithmetic Expression

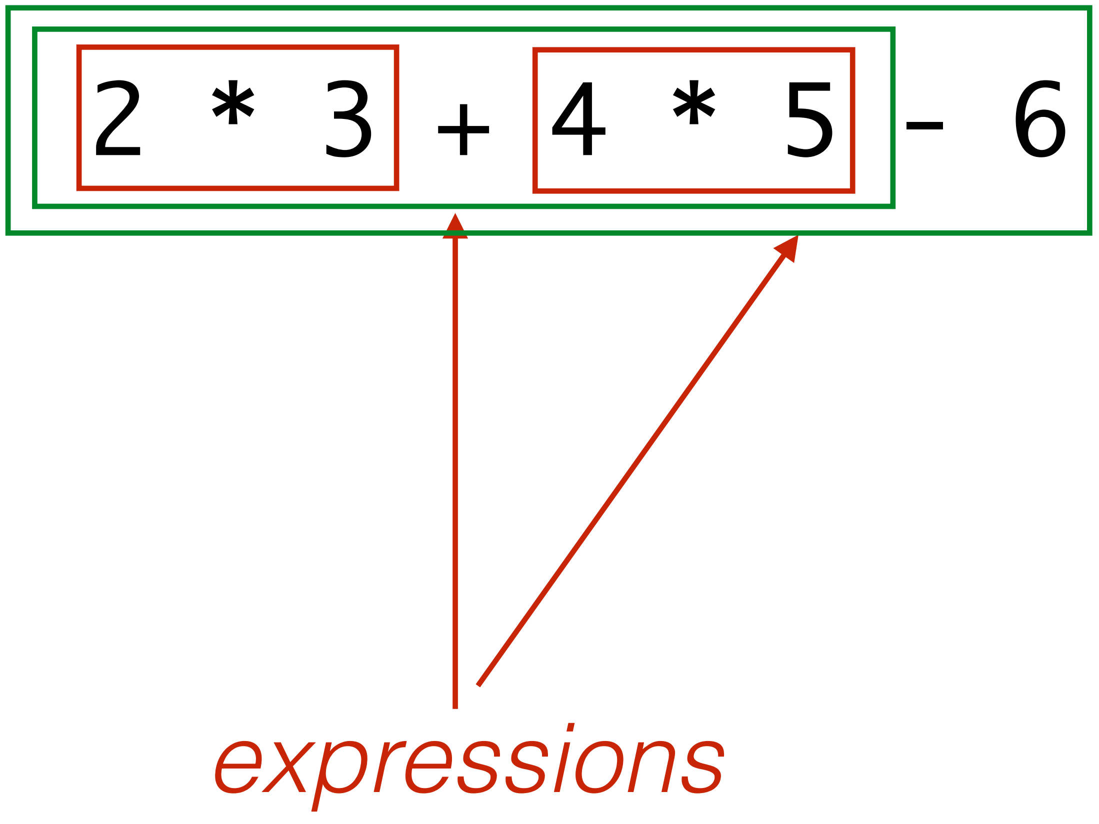


Arithmetic Expressions

```
  expr ::= term {“+” term | “-” term}.
  term ::= factor {“*” factor | “/” factor}.
factor ::= floatingPointNumber | “(” expr “)”.
```

Denotes one or more repetitions

Example Arithmetic Expression



This Grammar Encodes Operator Precedence

- Expressions contain terms
- Terms contain factors
- Factors only contain expressions if they are enclosed in parentheses

Encoding a Grammar Using Scala Parser Combinators

```scala
import scala.util.parsing.combinator._

class Arith extends JavaTokenParsers {
  def expr: Parser[Any] = term~rep("+"~term | "-"~term)
  def term: Parser[Any] = factor~rep("*"~factor | "/"~factor)
  def factor: Parser[Any] = floatingPointNumber | "("~expr~")"
}
```

Encoding a Grammar Using Scala Parser Combinators

```
import scala.util.parsing.combinator._

class Arith extends JavaTokenParsers {
  def expr: Parser[Any] = term~rep("+"~term | "-"~term)
  def term: Parser[Any] = factor~rep("*"~factor | "/"~factor)
  def factor: Parser[Any] = floatingPointNumber | "("~expr~")"
}
```

A parser for floating point numbers inherited from JavaTokenParsers.

Encoding a Grammar Using Scala Parser Combinators

```
import scala.util.parsing.combinator._

class Arith extends JavaTokenParsers {
  def expr: Parser[Any] = term~rep("+"~term | "-"~term)
  def term: Parser[Any] = factor~rep("*"~factor | "/"~factor)
  def factor: Parser[Any] = floatingPointNumber | "("~expr~")"
}
```

A combinator that takes two parsers and returns a new parser that first applies the left parser to its input, then its right to whatever remains.

Encoding a Grammar Using Scala Parser Combinators

```
import scala.util.parsing.combinator._

class Arith extends JavaTokenParsers {
  def expr: Parser[Any] = term~rep("+"~term | "-"~term)
  def term: Parser[Any] = factor~rep("*"~factor | "/"~factor)
  def factor: Parser[Any] = floatingPointNumber | "("~expr~")"
}
```

This combinator is overloaded so that string arguments are converted to simple parsers that match the string.

Encoding a Grammar Using Scala Parser Combinators

```
import scala.util.parsing.combinator._

class Arith extends JavaTokenParsers {
  def expr: Parser[Any] = term~rep("+"~term | "-"~term)
  def term: Parser[Any] = factor~rep("*"~factor | "/"~factor)
  def factor: Parser[Any] = floatingPointNumber | "("~expr~")"
}
```

A combinator that takes two parsers and returns a new parser that first applies the left parser to its input, and returns the result, unless the left parser fails (then it applies the right parser).

Encoding a Grammar Using Scala Parser Combinators

```
import scala.util.parsing.combinator._

class Arith extends JavaTokenParsers {
  def expr: Parser[Any] = term~rep("+"~term | "-"~term)
  def term: Parser[Any] = factor~rep("*"~factor | "/"~factor)
  def factor: Parser[Any] = floatingPointNumber | "("~expr~")"
}
```

A combinator that takes a parser and repeatedly applies it to the input as many times as possible.

To Convert a Grammar to a Definition with Parser Combinators

- Every production becomes a method
- The result of each method is `Parser[Any]`
- Insert the explicit operator ~ between two consecutive symbols of a production
- Represent repetition with calls to the function `rep` instead of `{ }`
- Represent repetitions with a separator with calls to the function `repsep`
- Represent optional occurrences with `opt` instead of `[ ]`

Exercising Our Parser

```
object ParseExpr extends Arith {
  def main(args: Array[String]) = {
    println("input: " + args(0))
    println(parseAll(expr, args(0)))
  }
}
```

An Example Parse of Grammatical Input

```
scala edu.rice.cs.comp311.lectures.lecture22.ParseExpr 2*3+4*5-6
input: 2*3+4*5-6
[1.10] parsed: ((2~List((*~3)))~List((+~(4~List((*~5)))), (-~(6~List()))))
```

An Example Parse of Ungrammatical Input

```
scala edu.rice.cs.comp311.lectures.lecture22.ParseExpr 2*3+4*5-6)
-bash: syntax error near unexpected token `)'
```

What is Returned from a Parser

- Parsers built from strings return the string (if it matches)
- ~ combinator returns both results
  - as elements of a case class named ~
  - (with a toString that places the ~ infix)
- `|` combinator returns the result of whichever succeeds
- `rep` operator returns a list of its results
- `opt` operator returns an `Option` of its result

Transforming the Output of a Parser

- The ^^ combinator transforms the result of a parser:
  - Let P be a parser that returns a result of type R
  - Let f be a function that takes an argument of type R

P^^f

  - Returns a parser that applies P, takes the result and applies f to it

Transforming the Output of a Parser

```
floatingPointNumber ^^ (_.toDouble)
```

Transforming the Output of a Parser

```
“true” ^^ (x => true)
```

Parsing JSON

- Many processes need to exchange complex data with other processes (often over a network)
- We need a portable way to represent the structure of data so that processes can conveniently send data amongst themselves
- One popular alternative is JSON
    - the Javascript Object Notation

Parsing JSON

- A JSON object is a sequence of members separated by commas and enclosed in braces
- Each member is a string/value pair, separated by a colon
- A JSON array is a sequence of values separated by commas and enclosed in square brackets

JSON Example

```
{
  “address book” : {
    “name” : “Eva Luate”,
    “address” : {
      “street” : “6100 Main St”
      “city” : “Houston TX”,
      “zip” : 77005
    },
    “phone numbers”: [
      “555 555-5555”,
      “555 555-6666”
    ]
  }
}
```

A Simple JSON Parser

```
class JSON extends JavaTokenParsers {
  def value: Parser[Any] = {
    obj | arr | stringLiteral |
      floatingPointNumber | "null" | "true" | "false"
  }
  def obj: Parser[Any] = "{"~repsep(member, ",")~"}"

  def arr: Parser[Any] = "["~repsep(value, ",")~"]"

  def member: Parser[Any] = stringLiteral~":"~value
}
```

Mapping JSON to Scala

- We would like to parse JSON objects into Scala objects as follows:
  - A JSON object is represented as a `Map[String,Any]`
  - A JSON array is represented as a `List[Any]`
  - A JSON string is represented as a `String`
  - A JSON numeric literal is represented as a `Double`
  - The values `true`, `false`, `null` are represented as corresponding Scala values