Comp 311
Functional Programming

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

October 31, 2019

Announcements

- Homework 3 is due before class next Thursday
- I’ll be grading the midterms this weekend

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 “)”.
```

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).

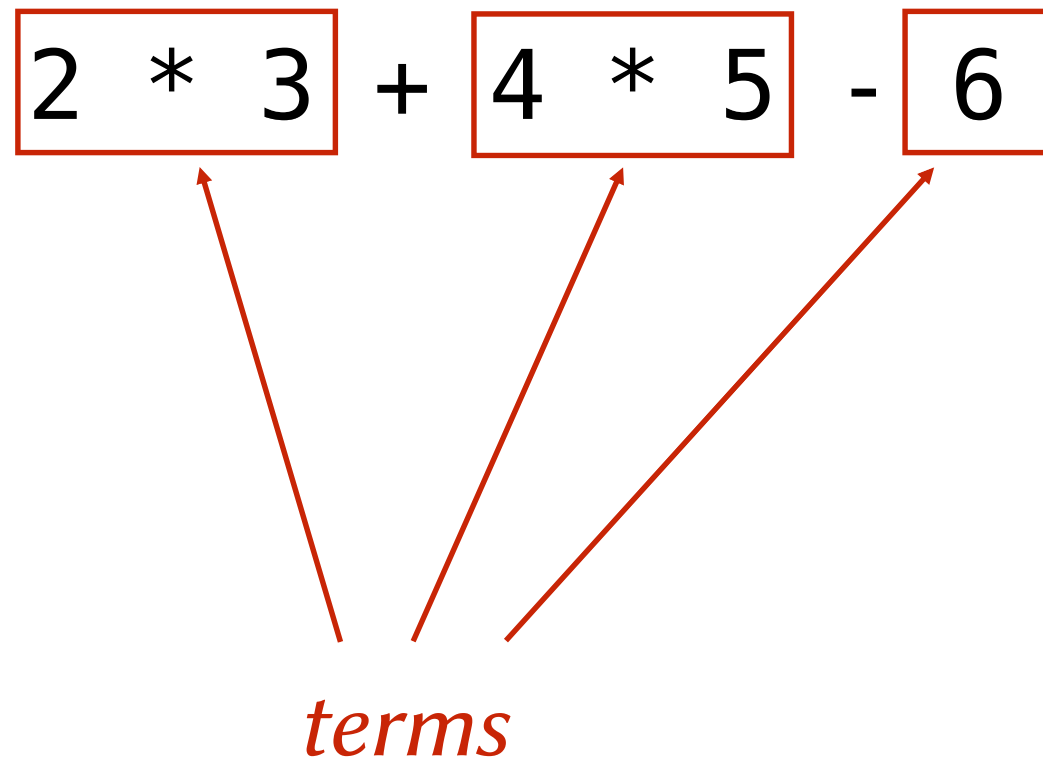
Example Arithmetic Expression

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

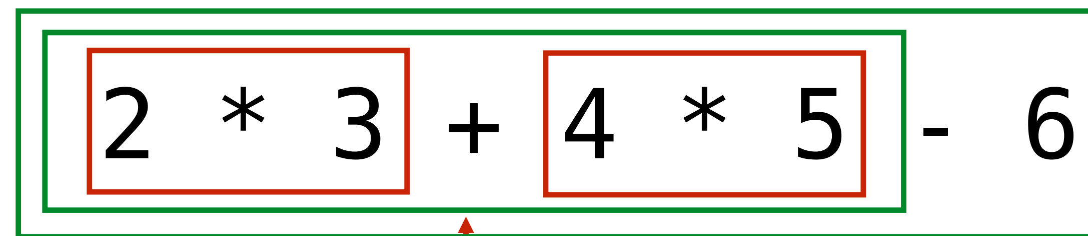
Example Arithmetic Expression

`2 * 3 + 4 * 5 - 6`

Example Arithmetic Expression



Example Arithmetic Expression



expressions

A Formal Grammar for Arithmetic Expressions in BNF

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

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

```
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

```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~")"
}
```

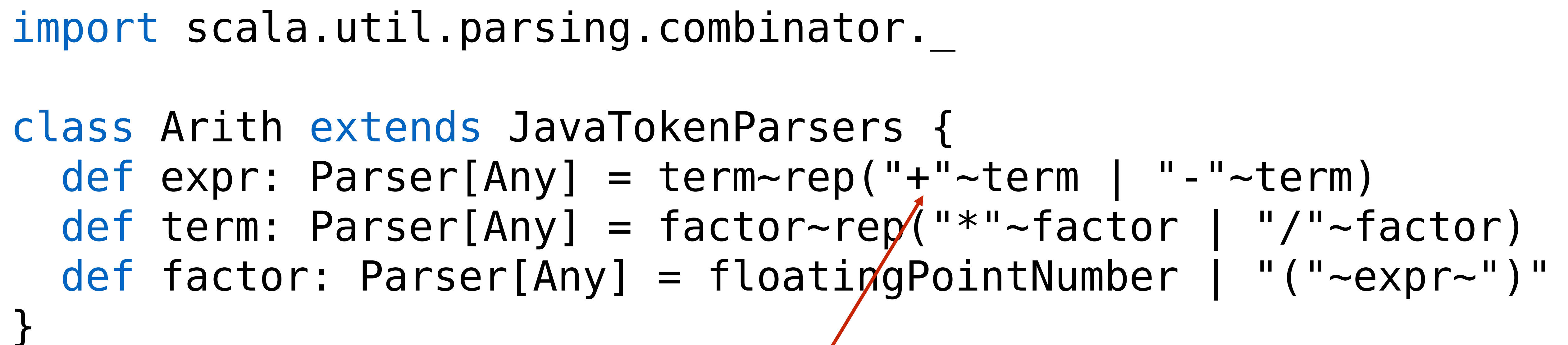
Encoding a Grammar Using Scala Parser Combinators

```
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 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

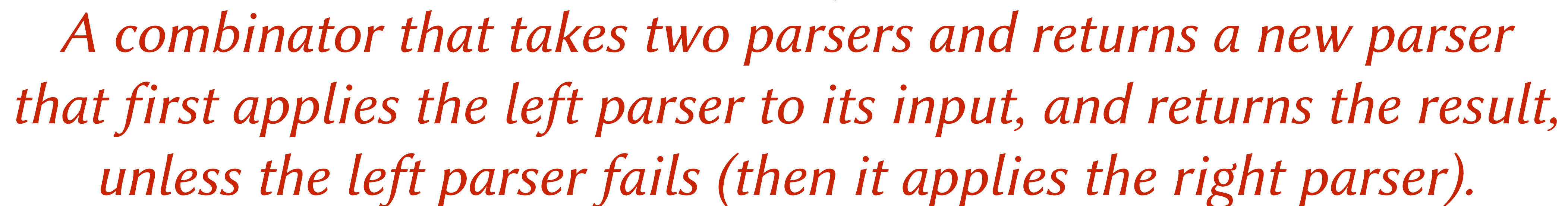


This combinator is overloaded so that string arguments are converted to simple parsers that match the string.

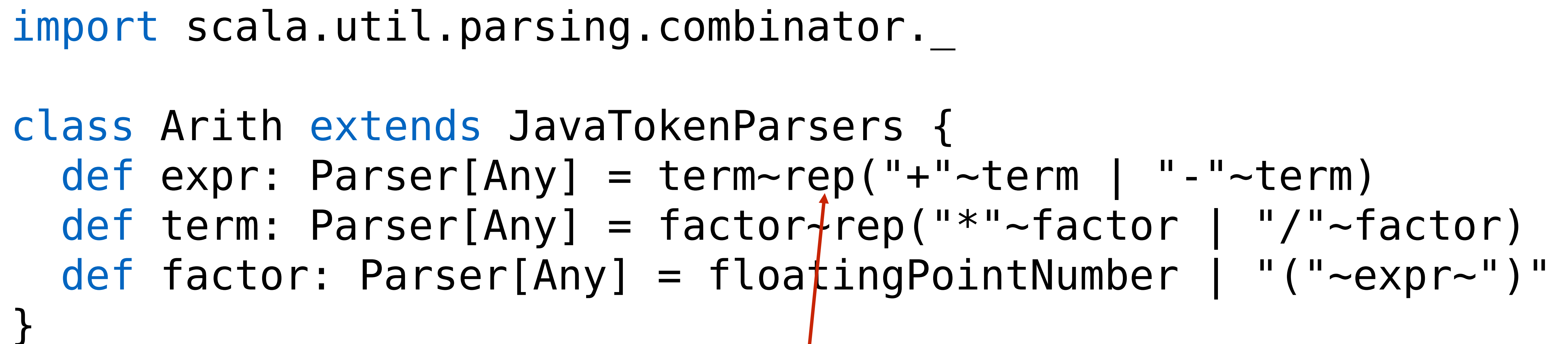
Encoding a Grammar Using Scala Parser Combinators

```
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~")"
}
```



Encoding a Grammar Using Scala Parser Combinators



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[T]`, where `T` is the expected result type (possibly `Any` if the result types have no better common supertype)
- Insert the explicit combinator-operator `~` between two consecutive symbols of a production to parse them in sequence
- Represent repetition with calls to the function `rep` instead of `{ }` (Note that the combinator `rep1` parses one or more repetitions)
- Represent repetitions with a separator with calls to the function `repsep` (Note that the combinator `rep1sep` parses one or more repetitions)
- 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` combinators return a list of results
- `opt` combinator 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)

“true” ^^^ 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

Definition of Class ~

```
case class ~[+A, + B](x: A, y: B) {
  override def toString = "(" + x + "~" + y + ")"
}
```

Redefining Member

```
def member: Parser[(String, Any)] = stringLiteral~":"~value ^^
{ case n~":"~v => (n,v) }
```

Redefining obj (Attempt 1)

```
def obj: Parser[Map[String, Any]] = "{"~repsep(member, ",")~"}" ^^
{ case "{"~ms~"}" => Map() ++ ms }
```

Redefining obj

- We can further improve our definition of obj by using the following parser combinators:

  ~> like ~ except that the left result is thrown out

  <~ like ~ except that the right result is thrown out

Redefining obj (Attempt 2)

```
def obj: Parser[Map[String, Any]] =
  "{"~>repsep(member, ",")<~"}" ^^ (Map() ++ _)
```

Complete JSON Parser with Mapping

```
class JSON2 extends JavaTokenParsers {
  def obj: Parser[Map[String, Any]] = "{"~>repsep(member, ",")<~"}" ^^
    (Map() ++ _)

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

  def member: Parser[(String, Any)] =
    stringLiteral~":"~value ^^
    { case n~":"~v => (n,v) }

  def value: Parser[Any] = {
    obj |
    arr |
    stringLiteral |
    floatingPointNumber ^^ (_.toDouble) |
    "null"  ^^^ null |
    "true"  ^^^ true |
    "false" ^^^ false
  }
}
```

Parsing a File

```
object JSONParseExpr extends JSON2 {
  def main(args: Array[String]) = {
    val f = Source.fromFile(args(0))
    try {
      println("input: " + args(0))
      println(parseAll(value, f.reader))
    }
    finally {
      f.close
    }
  }
}
```

Parsing a File

```
$ scala edu.rice.cs.comp311.lectures.lecture22.JSONParseExpr
sample.json
input: sample.json
[16.1] parsed: Map("address book" -> Map("name" -> "Eva Luate",
"address" -> Map("street" -> "6100 Main St", "city" -> "Houston TX",
"zip" -> 77005.0), "phone numbers" -> List("555 555-5555", "555 555-
6666")))
```