Comp 311
Functional Programming

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

Combinator Parsing

Arithmetic Expressions

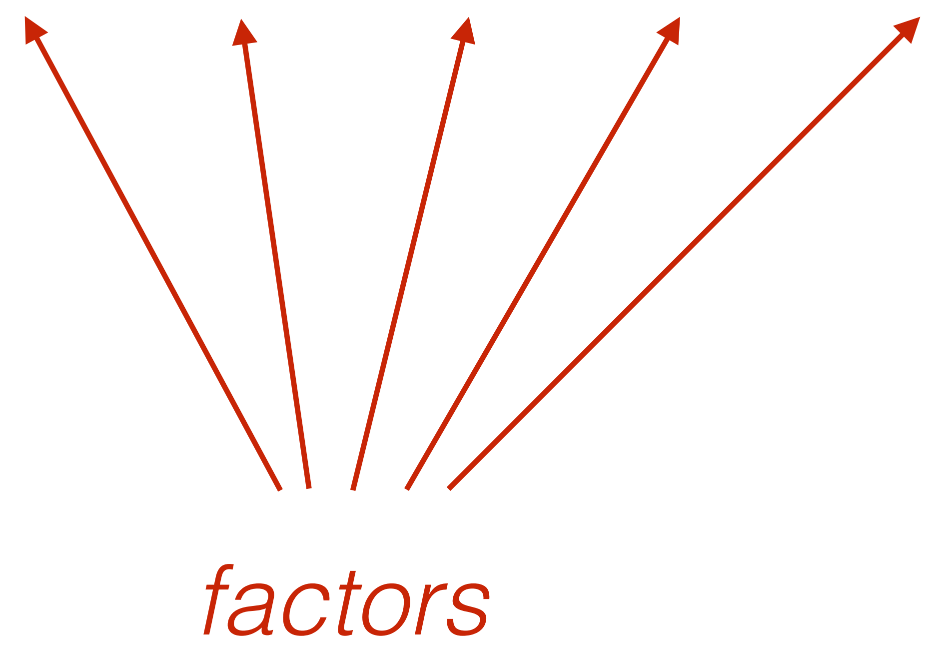
```
  expr ::= term {“+” term | “-” term}.
  term ::= factor {“*” factor | “/” factor}.
factor ::= floatingPointNumber | “(” expr “)”.
```

Example Arithmetic Expression

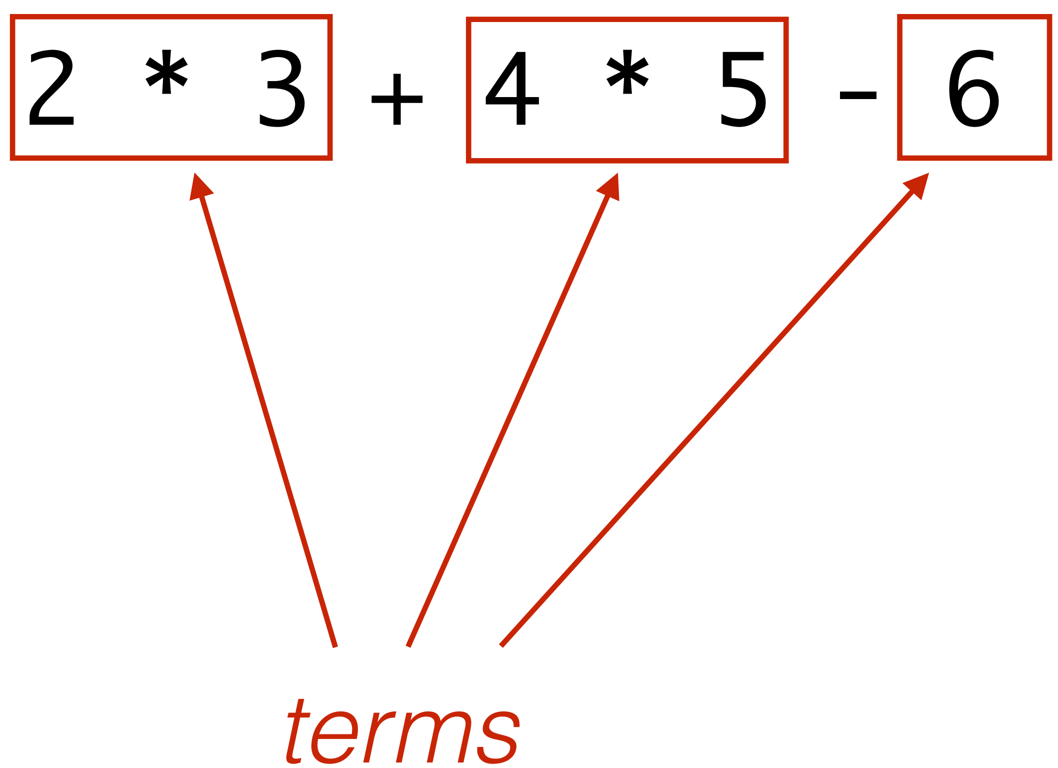
```
2 * 3 + 4 * 5 - 6
```

Example Arithmetic Expression

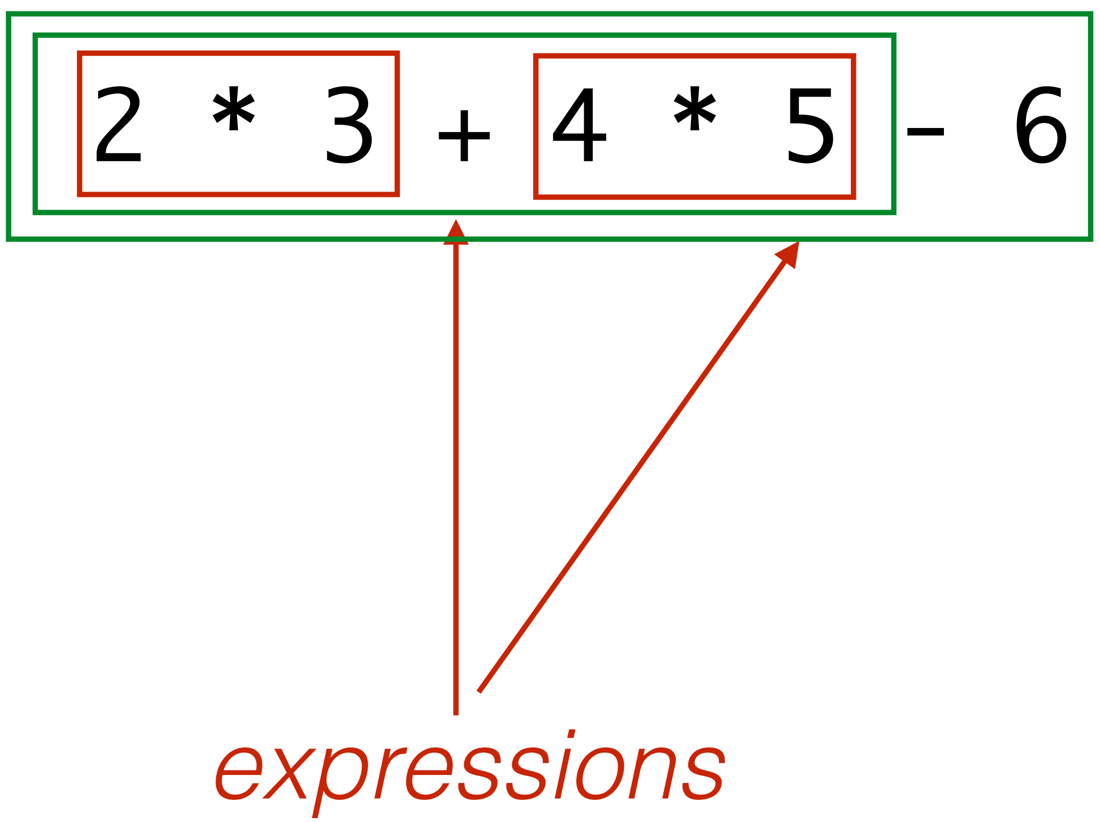
`2 * 3 + 4 * 5 - 6`



Example Arithmetic Expression



Example Arithmetic Expression



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~")"
}
```

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()))))
```

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

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

JSON grammar

```
  value ::= obj | arr | “null” | “true” | “false” |
            stringLiteral | floatingPointNumber
    obj ::= “{” [members] “}”
    arr ::= “[” [values] “]”
members ::= member {“,” member}
 member ::= stringLiteral “:” value
 values ::= value {“,” value}
```

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() ++ _)
```

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" ^^ (x => null) |
    "true" ^^ (x => true) |
    "false" ^^ (x => 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")))
```

Scala Actors and Concurrency

The Problem with Locks

- The JVM provides mechanisms for managing concurrent programs through *threads* and *locks*
- Programming with locks has many drawbacks:
  - Potential for deadlock
  - Locks at runtime are unknown
  - Threads at runtime are unknown

Scala Actors

- In Scala, concurrency is achieved through a *share-nothing* message passing model
- Actors are thread-like entities with mailboxes for receiving messages
- To implement an actor, extend `scala.actors.Actor`

A Simple Actor

```
import scala.actors._

object SimpleActor extends Actor {
  def act() {
    for (i <- 1 to 5) {
      println(“I’m acting!”)
      Thread.sleep(1000)
    }
  }
}
```

Starting Actors

- Actors are started by invoking their `start` method as with Java threads:

```
SimpleActor.start()
I'm acting!
res1: scala.actors.Actor = SimpleActor$@1945696

scala > I'm acting!
I'm acting!
I'm acting!
I'm acting!
```

Actors Run Independently

```
import scala.actors._

object ShakespeareanActor extends Actor {
  def act() {
    for (i <- 1 to 5) {
      println(“To be or not to be.”)
      Thread.sleep(1000)
    }
  }
}
```

Actors Run Independently

```
SimpleActor.start(); SeriousActor.start()
res2: scala.actors.Actor = seriousActor$@1689405

scala> To be or not to be.
I'm acting!
To be or not to be.
I'm acting!
To be or not to be.
I'm acting!
To be or not to be.
I'm acting!
To be or not to be.
I'm acting!
```

The actor Utility Method

```
scala> val shakespeareanActor2 = actor {
  for (i <- 1 to 5)
    println(“That is the question.”)
  Thread.sleep(1000)
}
scala> That is the question.
That is the question.
That is the question.
That is the question.
That is the question.
```

Actors Communicate Through Messages

- Send a message to an actor using `!`

```
SimpleActor ! “hello, simple actor”
```

Actors Communicate Through Messages

- Actors process the messages they receive using their `receive` method:

```
val echoActor = actor {
  while (true) {
    receive {
      case msg =>
        println(“received message: ” + msg)
    }
  }
}
```

Actors Communicate Through Messages

- When an actor sends a message, it does not block
- When an actor receives a message, it is not interrupted
- Actors ignore messages not handled in the function passed to receive

Actors Ignore Unmatched Messages

```
scala> val intActor = actor {
  receive {
    case x: Int => // I only want Ints
      println("Got an Int: " + x)
  }
}
```

Actors Ignore Unmatched Messages

```
intActor ! “hello”
intActor ! math.Pi
intActor ! 12
Got an Int: 12
```

Actors and Threads

- The Scala runtime manages one or more native threads for its use
- If you only work with actors you explicitly define, you do not need to worry about how actors map to threads
- You can view the current thread as an actor using `Actor.self`

Actor.self

```
scala> import scala.actors.Actor._
import scala.actors.Actor._

scala> self ! “hello”
scala> self.receive { case x => x }
res1: Any = hello
```

Actor.self

- When using the current thread as an actor, it is better to use `receiveWithin` (which takes a timeout) than `receive`
  - Especially if you are at the shell!

```
self.receiveWithin(1000) { case x => x }
res2: Any = TIMEOUT
```

Minimizing the Number of Threads

- Unfortunately, threads are expensive on JVMs
  - Thousands of threads vs millions of objects
  - Switching threads takes hundreds or even thousands of processor cycles
- Thus, for efficient programs, we want to minimize the number of threads

Receive vs React

- Along with receive, actors have a react method
  - Like receive, takes a partial function
  - Unlike receive, it never returns
    - Return type is `Nothing`

React Methods

- Because a react method never returns a value, it is not necessary to preserve the method's calling context
- Similar to tail calls:
  - With a tail call, the calling context is empty, so we need not preserve it
  - With react, the call never returns, so we need not preserve the calling context

React Methods

- By not preserving a calling context, we can reuse:
  - Space (the calling context)
  - Control (the calling thread)

React Methods

- Because a `react` method never returns:
  - It is responsible for performing all remaining computation of an actor
- Typically, this is done by having `react` call its actor's act method as its final action

```scala
object NameResolver extends Actor {
  import java.net.{InetAddress, UnknownHostException}

  def act() {
    react {
      case (name: String, actor: Actor) =>
        actor ! getIp(name)
        act()
      case “EXIT” =>
        println(“Name resolver exiting.”)
        // quit
      case msg =>
        println(“Unhandled message” + msg)
        act()
    }
  }
  def getIp(Name: String): Option[InetAddress] = {
    try { Some(InetAddress.getByName(name)) }
    catch { case _: UnknownHostException => None }
  }
}
```

React Methods and Loop

- Calling act as the last action of react can be made more concise with loop
- The loop function takes a thunk, calls the thunk, then calls itself, looping forever

Using Loop

```
def act() {
  loop {
    react {
      case (name: String, actor: Actor) =>
        actor ! getIp(name)
      case msg =>
        println("Unhandled message: " + msg)
    }
  }
}
```

Guidelines for Programming with Actors

Actors Should Not Block

- Design actors so that they do not block when processing messages:
  - If an actor blocks when processing a message, it will not notice other messages
  - If multiple actors block processing messages, waiting for other actors to respond, we can end up with circular waiting

Actors Should Not Block

- Instead of blocking, arrange for a message to arrive that indicates the action is ready to be taken
- It is ok to use a helper actor that does block waiting for an event (and does nothing else)
- This actor can then send an alert message to the actor it helps
- Because the helper receives no messages, it is ok to block

```scala
val emoter = actor {
  def emoteLater() {
    val mainActor = self
    actor {
      Thread.sleep(1000)
      mainActor ! “Emote”
    }
  }
  var emoted = 0
  emoteLater()

  loop {
    react {
      case “Emote” =>
        println(“I’m acting!”)
        emoted += 1
        if (emoted < 5)
          emoteLater()
      case msg =>
        println(“Received: ” + msg)
}}}
```

Non-Blocking Actors

- Because our example actor does not block, it is free to process other messages while waiting for the next emote message

```
scala> emoter ! “Hello”
scala> Receiver: hi there
I’m acting!
I’m acting!
I’m acting!
```

Communicate With Actors Only Via Messages

- The key advantage of the actor model is that we can reason about a multi-threaded program as a collection of single-threaded programs communicating via messages

- This advantage applies only if messages are the only way that actors communicate

Communicate With Actors Only Via Messages

- Do not call methods on actors explicitly — only send messages
    - Other methods might read or write private data, which would then be modified by multiple threads

Send Immutable Messages

- The data inside a message is shared by multiple actors
- It is best to make that data immutable to ensure thread safety
- An obvious way to accomplish this is to define methods using case classes
- Receive/react methods can easily process them with pattern matching

Make Messages Self-Contained

- When calling a function in a single-threaded context, a result is returned to the caller in the calling context
    - The caller “blocked” until the result was returned
    - It is easy for the caller to know what to do with the result

Make Messages Self-Contained

- With actors and message passing, the receiver is processing messages asynchronously
  - An actor might send a message to another actor and perform other work before it gets back a result (via another message)
  - It can be difficult for an actor to interpret the result messages it receives

Make Messages Self-Contained

- It helps to include in a message additional (even redundant) context to help the receiver interpret the message more easily
  - Define an abstract datatype with variants for each kind of message
  - Consider including the message being responded to

```scala
import scala.actors.Actor._
import java.net{InetAddress, UnknownHostException}

case class LookupIP(name: String, respondTo: Actor)
case class LookupResult (
  name: String,
  address: Option[InetAddress]
)

object NameResolver2 extends Actor {
  def act() {
    loop {
      react {
        case LookupIP(name, actor) =>
          actor ! LookupResult(name, getIp(name))
      }
    }
  }
  def getIp(name: String): Option[InetAddress] = {
    // as before
  }
}
```

Make Messages Self-Contained

- When calling a function in a single-threaded context, a result is returned to the caller in the calling context
    - The caller “blocks” until the result was returned
    - It is easy for the caller to know what to do with the result

Make Messages Self-Contained

- With actors and message passing, the receiver is processing messages asynchronously
  - An actor might send a message to another actor and perform other work before it gets back a result (via another message)
  - It can be difficult for an actor to interpret the result messages it receives

Make Messages Self-Contained

- It helps to include in a message additional (even redundant) context to help the receiver interpret the message more easily
    - Define an abstract datatype with variants for each kind of message
    - Consider including the message being responded to

```scala
import scala.actors.Actor._
import java.net{InetAddress, UnknownHostException}

case class LookupIP(name: String, respondTo: Actor)
case class LookupResult (
  name: String,
  address: Option[InetAddress]
)

object NameResolver2 extends Actor {
  def act() {
    loop {
      react {
        case LookupIP(name, actor) =>
          actor ! LookupResult(name, getIp(name))
      }
    }
  }
  def getIp(name: String): Option[InetAddress] = {
    // as before
  }
}
```

Scala Parallel Collections

Scala Collections Classes Include Parallel Counterparts

- `scala.collection.parallel.immutable`
  - `ParHashMap`
  - `ParHashSet`
  - `ParIterable`
  - `ParMap`
  - `ParRange`
  - `ParSeq`
  - `ParSet`
  - `ParVector`

Map and Flatmap

- These classes are intended to be constructed and used just like their sequential counterparts
- Because these classes implement map, flatmap, in parallel, for loops over them will execute in parallel
- A sequential collection can be converted into a parallel collection using the `par` method

Guidelines on Parallel Collections

- Benchmark use of parallel collections
  - Do not assume you will achieve speedup for a given program
  - Their benefit is most evident when the collections are large and we are mapping smaller, parallelizable operations over them